COVERAGE ANALYSIS FOR ASSERTIONS AND EMULATION BASED VERIFICATION

BY

VIRAJ MILIND ATHAVALE

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Assistant Professor Shobha Vasudevan

ABSTRACT

Coverage analysis is critical in pre-silicon verification of hardware designs for assessing the completeness of verification and identifying inadequately exercised areas of the design. It is widely integrated in the simulation based verification flow in the hardware industry. In this thesis, we provide solutions to enable effective coverage analysis in assertion based and emulation based verification.

We introduce two practical and effective code coverage metrics for assertions: one inspired by the test suite code coverage reported by Register Transfer Level (RTL) simulators and the other by assertion correctness in the context of formal verification. We present efficient algorithms to compute coverage with respect to the proposed metrics by analyzing the Control Flow Graph (CFG) constructed from the RTL source code. We apply our technique to a USB 2.0 design and an OpenRISC processor design and show that our coverage evaluation is efficient and scalable. We also present a technique to evaluate and rank automatically generated assertions based on fault coverage.

We present a novel technique to extract code coverage from emulation platforms. Using our CFG framework, we identify conditions or decision nodes and map them to other statements in the code. Triggering of decision nodes is recorded using additional trigger logic during emulation and mapped back to the source code to obtain coverage information. We apply our technique to an industrial design and show that it can efficiently provide fairly accurate code coverage statistics with minimal overheads during emulation.

To my family

ACKNOWLEDGMENTS

First of all, I would like to thank my adviser Professor Shobha Vasudevan, for her support and encouragement throughout my graduate work. She always believed in me and inspired me to take on new challenges. It is only because of her that graduate school has been such a great learning experience.

I would like to thank Sam Hertz, without whose help and insights the code coverage work would not have been possible. I would also like to thank Darshan Jaitly and Vijay Ganesan from Qualcomm for motivating the coverage work with a real world problem and for their ideas and constant support in exploring its various solutions. I would like to thank Dave Sheridan for his valuable help with GoldMine. Lastly, special thanks to Jayanand Asok Kumar, Lingyi Liu and Parth Sagdeo for the many research-related and other discussions we had that always helped and motivated me.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

RTL	Register Transfer Level
HDL	Hardware Description Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
ABV	Assertion Based Verification
HVL	Hardware Verification Language
FPGA	Field Programmable Gate Arrays
CFG	Control Flow Graph
LTL	Linear Temporal Logic
ILP	Integer Linear Program
GB	Gigabyte
RAM	Random Access memory
GHz	Gigahertz

CHAPTER 1

INTRODUCTION

1.1 Pre-Silicon Verification

With advances in semiconductor process technology, modern chips have become extremely complex, containing a few billion transistors and integrating a variety of hardware modules on a single die. Due to this ever increasing design complexity coupled with the huge costs associated with correcting hardware bugs, it is crucial to ensure design correctness prior to putting it in silicon. As a result, pre-silicon design verification today consumes the most computational resources and time in the chip development life cycle [1]. Its main objective is to ascertain that the design is behaving correctly, i.e. according to its specification. This objective is achieved by taking the design through various steps such as simulation or dynamic validation, formal verification, emulation and prototyping.

Most pre-silicon verification activities employ hardware designs described at the Register Transfer Level (RTL). At this level, Hardware Description Languages (HDLs) such as Verilog [2] and VHDL [3] are used to describe the design.

Simulation based verification involves running a set of tests[1] or a *test suite* through software simulation of RTL designs. The simulation output is checked against expected results with the help of variety of testbench checkers to determine whether the design is behaving correctly. The tests can be generated by pseudo-random test generators or are handwritten by the designers or verification engineers based on the functional specification of the design [4]. The latter are known as *directed tests*. Simulation based verification remains the most important component in the pre-silicon verification phase in the hardware industry [5].

Assertions represent desirable properties that a designs should satisfy. *Assertion Based Verification* (ABV) [6, 7] involves the use of assertions in formal verification as well as simulation to check if a design complies with its specification.

[1] Here we refer to pre-silicon verification stimuli and not post-manufacturing tests.

Formal verification statically analyzes the state space of the design to check if an assertion can ever be violated. When an assertion fails, formal verification tools provide a *counterexample* to show how the assertion can be violated. In simulation, assertions are continuously checked and an error is flagged when any assertion is violated. Synthesized assertions can also be used in emulation and silicon debug [8]. With increasing design complexity and the advent of modern Hardware Verification Languages (HVLs) such as Property Specification Language [9] and SystemVerilog [10], ABV has steadily gained popularity in recent years [11].

Emulation and prototyping platforms such as those based on Field Programmable Gate Arrays (FPGAs) have gained popularity to accelerate verification in recent years [12–17]. The primary motivation for using these platforms is speed: runtime of a test during emulation is orders of magnitude lower compared to simulation. For example, we found that for an industrial design, running a test in emulation was up to 360x faster than in simulation. The difference in runtimes becomes even more pronounced with increasing design complexity. The downside of emulation, however, is that it fails to provide complete visibility into the design.

1.2 Role of Coverage in Verification

Verification aims to ensure correctness of a design with respect to its specification. However, verification is only as effective as the test suite and the set of assertions used. For example, consider a test suite for a design containing a large number of tests, all exercising the `True` branch of an `if` condition in the design source code. Even if all the tests pass (produce correct output), the `False` branch is never exercised. Therefore, a bug existing in that portion of the code can escape the verification efforts. This leads to the following important questions: Is the test suite sufficient to ensure correctness? Have enough assertions been written and verified? How much of the design behavior has been verified? With ever increasing design complexity leading to enormous design state spaces, these questions are very difficult to answer. As a result, *coverage* achieved during verification plays a key role in determining the quality and completeness of verification results. Coverage analysis helps in quantifying verification completeness and identifying unexplored and untested parts of the design [5, 18, 19].

Each test in simulation based verification induces a different execution of the design, and a design can be guaranteed to be correct if it behaves as required for

all such possible tests. However, due to extremely high design complexity, only a subset all possible executions of a design can be checked in practice. Although the test suite is chosen so that the verification would be as exhaustive as possible, design errors in the untested parts of the design may still escape the verification process. As a result, measuring the exhaustiveness of the test suite becomes critical to judge verification progress. Coverage analysis proves to be an effective tool for this task.

A related problem in formal verification is to measure the exhaustiveness or completeness of specification as defined by the set of assertions. An erroneous design behavior can escape the formal checks if it is not captured by the specification. Indeed in the majority of cases, the specification is manually written by a designer, due to which its quality depends on the competence of the designer. Determining the behaviors covered by the specification is therefore indispensable in this case as well.

Coverage of a test suite or assertions is a broad term encompassing different aspects of the design that are verified. For example, it may correspond to the parts of the design source code exercised by a test suite, or to the fraction of design states or state transitions. It may even correspond directly to important design behaviors as identified by a designer. This has led to a wide variety of *coverage metrics* often orthogonal to each other [18]. We survey the various metrics in Section 1.3.

Coverage analysis not only helps in quantifying the progress of various verification activities, but can identify inadequately verified design aspects as well. Identification of such coverage *holes* guides future verification efforts in the right direction. Additional tests or assertions can then be written to fill these holes. Such a coverage guided approach leads to more systematic verification efforts and ensures optimal use of resources. From a more philosophical point of view, coverage analysis provides an answer to the fundamental question: Does the design meet its specification completely? The ultimate goal is to achieve *coverage closure*, i.e. to exhaustively verify the entire design behavior.

1.3 Coverage Metrics

1.3.1 Coverage Metrics for a Test Suite

For conventional simulation based verification, a wide variety of coverage metrics have been proposed and are currently in use in the hardware industry. The coverage metrics for a test suite can be broadly classified into six categories as follows [5, 18].

- **Code coverage metrics:** Code coverage identifies the different **structure classes of the HDL program** for the design that are exercised during simulation. The structure classes considered are statements, branches and expressions, each giving rise to a different metric. Code coverage metrics are largely derived from software testing [20]. Since code coverage can be easily related to the source code, it is relatively easier to fill the identified coverage holes. Additionally, measuring code coverage adds little overhead to simulation. Therefore, these are the most popular coverage metrics and serve as a key sign-off criteria for ASIC tapeout in the hardware industry. However, due to the inherent concurrency present in hardware designs, achieving 100% code coverage does not guarantee complete functional correctness and more complex coverage metrics are required.

- **Metrics based on circuit structure:** These metrics refer to the circuit that describes the design and identify **physical parts of the circuit** (e.g. latches) that are not exercised during simulation [21]. Measuring coverage w.r.t. these metrics is easy; however, it is much more difficult to write tests to fill the coverage holes. Eliminating false negatives in this case is also a challenging task [5].

- **Finite state machine based metrics:** Finite State Machine (FSM) metrics require **state, transition or limited path coverage** on an FSM description of the design. Analyzing such a description for the full system is rarely feasible; hence, these metrics are applied to smaller abstract FSMs either hand-written by designers [21, 22] or automatically extracted from the design description [23]. Metrics in this category are superior to the code or circuit based coverage metrics due to their ability to capture, albeit partially,

the sequential behavior of the design. However, it is not straightforward to interpret the coverage reports and write tests to improve coverage.

- **Functional coverage metrics:** For this class of metrics, a list of **error-prone scenarios or functionality fragments** (also known as *coverpoints*) is manually constructed by verification engineers or automatically extracted from the RTL description of the design [24]. Coverage is computed based on the fraction of the coverpoints exercised during simulation. Identifying such coverpoints, however, is a laborious manual task requiring in-depth understanding of the design. Since coverpoints can be thought of as assertions, functional coverage is also referred to as *assertion coverage*, not to be confused with the coverage of the assertions themselves.

- **Observability coverage metrics:** The coverage categories described thus far are based on the *activation* of different aspects of the design during simulation. These collectively determine *controllability coverage* of verification [25]. On the other hand, *observability coverage* checks if the effects of errors activated by tests can be **observed at the design outputs** [26, 27]. These are relatively recent coverage metrics.

- **Fault coverage metrics:** Fault (error) coverage analysis involves **injecting faults into the design according to different models** and checking if it causes erroneous behavior [28]. The fault coverage of a test denotes the fraction of faults for which the test fails, or *catches* the fault. These metrics model a design error by a local mutation in a design description format such as a net list, HDL code fragment, or state transition diagram. They are inspired from software testing [29] and post-manufacturing testing of hardware [30].

Table 1.1 summarizes the most widely used coverage metrics for pre-silicon verification in the hardware industry.

Determination of the coverage of a test suite when it is applied to a design being emulated in a "black box" emulation environment is a daunting practical challenge. Standard emulation platforms and tools are unsuitable for measuring code coverage. In spite of this, some complex tests are run on emulators alone, due to extremely long simulation runtimes. Currently, such tests are only used to check functional correctness of the design but not for coverage. Since emulation

Table 1.1: Summary of important coverage metrics for a test suite. For more details and comparison between the different metrics, see [5] and [31].

Coverage metric	**Coverage category**	**Description**
Statement coverage	Code	Statements executed during simulation.
Branch coverage	Code	Branches (`if, case`) that are completely exercised.
Expression coverage	Code	Expressions for which all possible ways of evaluating it to each possible value are exercised.
Toggle coverage	Circuit	Counts the number of times a circuit node *toggles*, i.e. changes from 0 to 1 or 1 to 0.
State and transition coverage	FSM	Measures the number of times each state and state transition is exercised.
Functional coverage	Functional	Records whether each manually specified error-prone scenario and functionality fragment is exercised.

is able to run tests in a fraction of the time required for simulation, it should be leveraged for coverage analysis as well.

1.3.2 Coverage Metrics for Assertions

Traditionally, coverage metrics for assertions (properties) in the context of formal verification reason about the FSM model of a design [32–35]. Essentially, coverage is measured in terms of the number of states in the FSM model covered by the assertions. The pioneering approaches of Hoskote et al. [32] and Katz et al. [33] propose two distinct directions in defining and computing the state space based coverage. The approach in [32] is based on mutations applied to the FSM. A state is covered by a property if modifying the value of a variable in the that state causes the property to fail formal verification. Katz et al. [33] define coverage based on a comparison between the FSM and a reduced tableau of the property.

Fault or mutation coverage metrics have also been proposed for properties [36, 37]. These metrics use the number of faults or mutations detected as a notion of coverage.

Modern verification flows integrate conventional simulation based verification and formal property verification [21]. In order to get a unified picture of coverage

achieved during verification, it is valuable to define assertion coverage metrics on the lines of test suite coverage metrics and provide efficient methods to compute them. Similarly to test suites, such analysis can direct verification engineers to inadequately verified parts of the design for which more assertions need to be written. Chockler et al. [18] present an important step in this direction by providing new coverage metrics in the context of formal verification corresponding to the ones used for a test suite. The support provided by commercial tools in this direction is limited (exceptions include [38, 39]).

Apart from judging verification completeness, coverage analysis of assertions is essential to evaluate and rank the assertions themselves. This can help in identifying a small set of high-quality assertions that can then be used in various stages of the design cycle including simulation, formal verification, emulation and Silicon debug. Such ranking is particularly valuable when assertions are automatically generated [40].

1.4 Proposed Solutions for Coverage Analysis

In this work, we address the issues identified in Section 1.3 for coverage analysis in the context of emulation and assertion based verification and provide effective solutions for each of them.

1.4.1 Coverage in Assertion Based Verification

Firstly, we define two practical and effective code coverage metrics for assertions, one for each setting in which assertions are primarily used: simulation and formal verification. The *simulation based coverage metric* defines the coverage of an assertion in terms of the code coverage of tests that trigger that assertion. The *correctness based coverage metric* for an assertion focuses on the statements that affect the correctness of that assertion. We thus focus on *statement coverage* of assertions.

We present algorithms to compute coverage according to the proposed coverage metrics. The algorithms rely on statically analyzing Verilog RTL source code for the design as a *Verilog program* [41, 42]. To facilitate this analysis, the design source code is represented as a Control Flow Graph (CFG) extended with the

information about variable dependencies which is required for coverage computation.

Our coverage definition and computation technique has key merits over the state space based methods. Firstly, the constructed CFG is linear in the size of RTL source code. Therefore, the cost of building a CFG is far less than a state transition graph, and any manipulations of the CFG are also more scalable. Secondly, coverage reported in terms of lines of RTL source code is closer to the designer and easier to interpret. On the other hand, state space coverage is not easily translatable to source code. Lastly, in practical verification environments, bridging coverage holes in an assertion suite can be easier when coverage information is available in the form of lines of code. Since verification is a resource and time intensive process, it is valuable to make coverage information easy to understand and use. In addition, our technique can help in quickly determining how modifications to an assertion suite can affect coverage.

We evaluate our correctness based coverage metric and analysis technique using a USB 2.0 protocol design and OpenRISC processor design from [43]. We inject mutations into the covered statements and see if the assertion fails formal verification in the mutated design. These experiments help show that the technique is scalable and efficient and correctly computes coverage according to the proposed definition.

Finally, we study the use of fault coverage, or the number of faults detected by assertions, as a metric to evaluate the assertions automatically generated by the GoldMine tool [40, 44]. We model Single Event Upset (SEU) faults in the design through mutation of the design source code. If an assertion is formally verified true in the original *fault-free* design but fails formal verification in the *faulty* design, it is said to detect the injected fault. Moreover, the number of assertions detecting a particular fault can be used to estimate the criticality of that fault w.r.t. design outputs.

We present a formal analysis to explain how GoldMine assertions detect the injected SEUs. We also show the effectiveness of our approach through experiments on the SpaceWire communication controller design.

1.4.2 Coverage in Emulation Based Verification

We apply our simulation based code coverage computation technique for assertions to address the problem of coverage extraction from emulation and prototyping platforms. For each conditional statement in the source code, we identify a corresponding set of statements using static analysis. These statements are such that during running of a test, the condition evaluating to true during running of a test ensures execution of the set of statements. Extra logic is added to the design to record the evaluation of these conditions during emulation. The recorded information is post-processed to find the statements executed or *covered* during the emulation. In order to limit the area overheads of the additional logic, the set of conditions is optimized prior to emulation.

Our technique is targeted towards computing the statement coverage metric for a test. Branch coverage can also be derived from the output of our technique. Moreover, it should be noted that although we mainly consider FPGA based emulation platforms, the technique can be readily applied to other hardware emulators. This is due to the fact that the trigger logic to be added is expressed as synthesizable Verilog code.

We demonstrate our technique on an industrial design emulated on a Xilinx FPGA platform. The runtime of our technique was within 30 seconds for the design with a few thousand lines of source code, which shows that the technique is scalable and efficient. The code coverage reported using our technique is compared to that from an RTL simulator. We find that with around 10% FPGA usage overhead, the error in reported coverage by our technique is within 2% on average. Moreover, using optimization, the overhead can be reduced at the cost of slightly higher error.

1.5 Summary

In summary, the main contributions of this thesis are as follows.

- We propose two code coverage metrics for assertions, applicable in each of the two use cases: simulation and formal verification.
- We present an efficient technique to compute code coverage of assertions by statically analyzing the RTL source code. Our technique presents statement

coverage, which is easier to interpret for the designer as compared to state space coverage.

- We provide an objective evaluation of automatically generated assertions from the GoldMine tool based on fault detection and show how the same analysis can also be employed to estimate criticality of faults.

- We provide a practical and efficient solution to the problem of extracting code coverage from emulation and prototyping platforms using static analysis of design source code.

The rest of the thesis is organized as follows. Chapter 2 discusses previous work related to the different components of the thesis. In Chapter 3, we describe our approach of code coverage analysis for assertions. It includes definitions of the proposed code coverage metrics, our CFG framework and the coverage computation algorithms that utilize the framework. Chapter 4 explains our approach of ranking assertions based on fault detection and estimating criticality of faults. Our technique of code coverage extraction from emulation and prototyping platforms is next explained in Chapter 5. Through a motivating example, we show how CFG based static analysis can help extract code coverage efficiently. This is followed by a detailed description of our methodology and experimental results on an industrial design. We conclude in Chapter 7.

CHAPTER 2

RELATED WORK

We consider previous work related to various parts of the thesis including coverage in verification, assertions and coverage analysis for assertions, fault criticality evaluation, coverage during emulation and prototyping, and static analysis.

2.1 Coverage in Verification

As explained in Chapter 1, coverage analysis is indispensable to evaluate the exhaustiveness and quality of verification of hardware designs. Therefore coverage metrics and coverage analysis techniques have been widely studied [5, 18, 19] and incorporated in industrial design flows [4, 21, 24].

Tasiran and Keutzer [5] survey the myriad of coverage metrics for test suites currently employed in simulation based verification. Code coverage metrics [31, 45] such as statement coverage and branch coverage refer to parts of the HDL source for the design exercised during simulation. Metrics such as toggle coverage identify physical parts of the circuit such as latches that are not fully exercised [21]. FSM metrics measure coverage in terms of states and state transitions of an abstract FSM model of the design. The model can be either hand-written by designers [21, 22] or automatically extracted from the design description [23, 46–48]. Hoskote et al. [23] present a technique of automatically extracting the control flow of a design on the basis of the underlying mathematical model, independent of the circuit description style. Other approaches [30, 48] select state variables for the automatically extracted FSM based on architectural information. Functional coverage involves checking if manually or automatically identified error scenarios are covered during simulation [24]. More recent metrics [26, 27] take into account observability information to determine whether effects of errors activated by a test can be observed at the circuit outputs. Metrics based on fault models injecting faults into the design and checking whether it causes erroneous behavior [28].

Code and circuit coverage statistics are readily reported by most commercial RTL simulators [49–51]. Code coverage analysis is done through instrumentation based or dumpfile based techniques [31]. Instrumentation based techniques utilize the Programming Language Interface (PLI) of simulators to measure the execution statistics of the source code during simulation. Dumpfile based techniques compute coverage statistics by post-processing of Value Change Dump (VCD) files generated during simulation. Among the two methods, dumpfile based methods incur much lower performance overhead during simulation, and different coverage metrics can easily be computed from the same dumpfiles.

Code, mutation and even functional coverage metrics are used in software testing to evaluate the exhaustiveness of the test suite [20, 52, 53]. Zhu et al. [20] survey the extensive research in this area. The code coverage metrics for simulation based hardware verification are largely derived from those used in software testing.

In Integrated Circuit (IC) manufacturing testing, fault coverage is used as a metric to evaluate the effectiveness of test patterns to isolate a defective chip [54, 55]. Guo et al. [56] compare the various fault models, e.g. stuck-at fault, bridging fault and N-defect, used in post-manufacturing testing through analysis of large amounts of production test data. Some of these fault models such as stuck-at fault are also used in the context of pre-silicon verification. Moundanos et al. [30] propose a unified framework for verification and post-manufacturing testing. It is shown that the same abstraction techniques can aid Automatic Test Pattern Generation (ATPG) tools to attack hard-to-detect faults as well as provide a meaningful measure of coverage for design verification.

2.2 Assertions

The idea of an assertion or property for checking correctness can be traced back to Turing [57], where an approach of partitioning a large verification problem into a set of assertions was first presented. Use of assertions to formally specify and reason about programs was proposed by Floyd [58] and Hoare [59]. The onset of HDLs first introduced assertions in the context of hardware design verification. For instance, VHDL [3] '`assert`' keyword provides a way of expressing a condition as property and checking if it is ever violated during execution. Model checking [60] was proposed around the same time to formally check the

correctness of a design w.r.t. specifications expressed in the form of assertions. Modern verification processes involve the use of powerful HVLs such as PSL [9] and SystemVerilog [10] that allow expressing complex assertions. Many commercial tools now provide support for assertion based verification, where assertions expressed in these HVLs are used in conjunction with simulation and formal verification [6, 7].

Writing assertions manually is a very difficult and laborious task. Various automatic assertion generation techniques have been proposed for software [61–64] and hardware [40, 65–67] to alleviate this problem. These techniques use a dynamic or static analysis approach or their combination for assertion generation.

Early work with software assertion generation focused on extracting *loop invariants* or properties that hold across iterations of a `while` or `for` loop in the program [68]. DIDUCE [61] is an online technique to learn invariants on variables in the program at specific program points. Ammons et al. [63] present a machine learning approach to discovering formal specifications of programs. Daikon [62] discovers invariants by analyzing program execution traces and inferring invariants based on predefined templates such as comparison with constant value, linear relationships and ordering.

The IODINE tool [65] employs a template based dynamic approach similar to Daikon to generate assertions for hardware. The templates used include one-hot, mutual exclusion and request-acknowledge. A sequential data mining approach to extract hardware assertions is proposed in [66]. GoldMine [40] uses a novel combination of data mining and static analysis to generate assertions. The formal verification step in GoldMine ensures that the generated assertions are true system invariants. In this thesis, we evaluate and rank GoldMine assertions on the basis of fault coverage.

2.3 Coverage Analysis for Assertions

Assertion or property coverage in the context of formal verification has been traditionally expressed in terms of the fraction of design states covered [32–35]. Coverage computation in this case typically depends on the analysis of the state transition graph of the design or its variants. Hoskote et al. [32] present a mutation based approach where a state is covered by a property if modifying the value of a variable in that state causes the property to fail formal verification. This work

was followed up in [34, 35]. Katz et al. [33] propose the use of a comparison between the FSM and a reduced tableau of the property to define coverage. In our coverage analysis technique, we do not construct a state transition graph from the RTL design, but instead perform an analysis of the Verilog Hardware Description Language (HDL) source code for the design considered as a *Verilog program* as in [41, 42].

Recent approaches [18, 69] attempt to relate coverage metrics from formal and simulation based verification. For each of the simulation based coverage metrics, [18] presents a corresponding metric suitable for assertions in formal verification. To compute coverage of an assertion, statements in the code are removed one at a time, and the assertion is checked for vacuity in the mutant design. Although a pioneering approach to code coverage of assertions, it becomes intractable as size of the design source increases. In contrast, our technique does not require mutating every line and, since we analyze the CFG, the technique naturally scales. In [69], the authors propose the use of a test plan language as a formal basis for unifying the coverage goals for simulation and formal property verification. However, the approach does not provide a way to map assertions to the design source.

Some commercial formal verification tools have recently started providing code coverage statistics for assertions in the context of formal verification through analysis of the state transition graph of the design [38,39]. We provide different coverage metrics for assertions applicable in simulation and formal verification. Based on source code analysis, we also give efficient techniques of coverage computation which are inherently more scalable.

2.4 Fault Coverage and Criticality Evaluation

Simulation based approaches are typically used for fault experiments to determine fault coverage of assertions [70]. Since our fault coverage evaluation method uses formal techniques for all fault experiments, it accurately identifies the faults detected by assertions. In [71, 72], formal methods have been used to inject soft errors in RTL design and evaluate fault coverage, but the authors only consider manually written assertions. We use an approach similar to [71] to evaluate the automatically generated assertions from the GoldMine tool [40].

Our SEU fault criticality analysis can be thought of as a part of Soft Error

Rate (SER) determination for digital circuits, which is widely studied at circuit, gate and architectural levels [73–78]. Most approaches are simulation based and study the probability of a transient fault at a circuit node getting latched in a flip-flop/latch or the effect of soft errors visible at the architectural level. We define the criticality of an SEU fault at a flip-flop/latch in terms of the number of paths through which it can propagate to a design module output and use formal methods for criticality estimation. In [73], an analytical method is presented to analyze multi-cycle error propagation behavior to find fault criticality. Although the method accurately computes propagation probabilities, the analysis is performed at gate level. We reuse automatically generated RTL assertions for criticality estimation.

2.5 Emulation and Prototyping in Verification and Coverage

Use of emulation and prototyping platforms such as those based on Field Programmable Gate Arrays (FPGAs) to accelerate verification in the hardware industry is widely studied [12–14]. Ray and Hoe [13] present a case study in developing a synthesizable high-level model of a superscalar processor and producing a working prototype in FPGA. Emulation and prototyping is now commonly integrated in industrial verification flows [15–17]. For instance, Gateley et al. [16] discuss the experiences in applying emulation based techniques to aid design verification of the UltraSPARC-I processor.

To the best of our knowledge, no previous approaches exist for extracting code coverage from emulation platforms and which apply static analysis for this purpose. The VN-Cover Emulator tool [79] enables code coverage extraction from some emulation platforms such as Cadence Palladium and Cobalt using code instrumentation similar to coverage analysis by RTL simulators. Through the CFG based analysis and decision node optimization, our technique for coverage extraction from emulation efficiently provides coverage statistics with minimal overheads.

2.6 Static Analysis and Control Flow Graphs

In software, static analysis encompasses a family of techniques for automatically computing information about the behavior of a program without executing it [80]. Static analysis techniques are used in a variety of fields including compiler optimization [81], debugging [82], security [83], formal verification [80] and invariant generation [64]. Many of these construct a Control Flow Graph (CFG) or its variants (e.g. Program Dependence Graph (PDG) [81]) from the program to facilitate the program analysis. *Program slicing* [84], a static analysis technique that identifies program statements relevant to a particular computation, also uses CFGs.

High-level synthesis converts a system-level behavioral description into an RTL description optimized for energy, power or area [85]. Control Data Flow Graphs (CDFG) commonly serve as the intermediate representation of the system in this case.

Static program analysis techniques have been adapted for hardware by treating the HDL source for a design as a program [41,42]. Clarke et al. [41] propose using a PDG [81] to represent each of the concurrent processes in the HDL source. The PDG captures both control and data dependencies and thus is an extension of a CFG. We use an extended CFG framework similar to a PDG for our code coverage analysis techniques.

CHAPTER 3

CODE COVERAGE ANALYSIS OF ASSERTIONS

3.1 Introduction

In this chapter, we present our work on code coverage analysis of assertions. In particular, we consider statements coverage for assertions. We start by describing our Control Flow Graph (CFG) framework to facilitate static analysis of Verilog source code as a program [41] (Section 3.2). We then define *simulation based* and *correctness based* code coverage metrics relevant to each of the two use cases of assertions, viz. simulation and formal verification (Section 3.3). Using the CFG framework, we present algorithms to compute coverage according to each of the proposed metrics (Sections 3.4 and 3.5). We demonstrate experimental results on a USB protocol design and the OpenRISC processor design in Section 5.4.

Our CFG framework consists of a CFG for each of the concurrent processes in the design source. The CFG is further augmented with data flow information in the form of variable dependencies. CFG for the design consists of the union of CFGs for all processes in each of the design modules. Statements in the code correspond to CFG nodes while branches correspond to edges. Thus, the CFG is linear in the size of the design source.

Among the proposed coverage metrics, *simulation based coverage metric* defines the coverage of an assertion to closely approximate the coverage of the tests that trigger that assertion. On the other hand, the *correctness based coverage metric* focuses on the statements that affect the correctness of that assertion.

In simulation based coverage computation, we identify the set of statements such that triggering of the assertion (i.e. its antecedent evaluating to true) ensures execution of these statements. This set consists of the statements that must be executed for the decision node to trigger (*backward cone*) and the statements conditioned on the decision node (*forward cone*). The analysis is performed in a way that the coverage from our technique closely approximates the code coverage

reported by a simulator.

Correctness based coverage computation consists of an execution phase and a coverage extraction phase. In the execution phase, assuming the antecedent of the assertion holds, the CFG is used to explore all possible executions of the RTL design over the number of cycles spanned by the assertion. During the CFG traversal, important dependency information between statements is stored in the form of *triggers*. These triggers are used in the coverage extraction phase to find statements covered by the assertion.

We evaluate our correctness based coverage metric and analysis technique using a USB 2.0 protocol design and OpenRISC processor design from [43]. We find that on average, only about 8% of the total statements were reported as covered by each assertion, which shows that our technique can effectively find the statements truly relevant to the correctness of an assertion. We inject mutations in the covered statements and see if the assertion fails formal verification in the mutated design. These experiments help show that the technique correctly computes coverage according to the proposed definition. For every assertion, mutations at almost all covered statements were detected. Careful analysis of the source code with the assertion shows that the mutations are not detected only in cases listed in Section 3.5.

The simulation based coverage analysis is evaluated by applying it to solve the problem of code coverage extraction from emulation and prototyping platforms.

3.2 Verilog RTL Source Code as a Control Flow Graph

In order to facilitate analysis of the Verilog source code, we represent it as a *Control Flow Graph (CFG)*. Each statement in the code is mapped to a node in the CFG. A design is made up of a set of modules. Each module in turn consists of a set of concurrently running `always` processes and `assign` statements. The CFG for the module consists of the union of CFGs for each of these processes. The set of CFGs for all modules thus completely captures the structure of the design. It is similar to a Process Dependence Graph described in [41].

Each node in the CFG stores the number of the line in the RTL code it corresponds to, as well as an expression for the RTL statement at that line.

An *assignment node* represents a blocking or non-blocking assignment in the

Verilog code such as '`a = 2'b01`' or '`b <= 1'b1`'. A *decision node* represents conditional statements, i.e. `if` and `case` statements in the source code. Nodes other than assignment and decision nodes correspond to `begin`, `end`, `always @ ()`, etc.

Edges in the CFG represent the control flow between RTL statements. Each assignment node has one outgoing edge that points to the node corresponding to the next line in the RTL code. Each decision node has two outgoing edges $left$ and $right$ that point to the RTL statements executed if the corresponding condition is true or false, respectively.

The CFG is a purely syntactic representation of the RTL code. For effective coverage estimation of assertions, we need to track dependencies between variables. The CFG is therefore extended with additional data flow information. The resulting *extended CFG* is similar to the System Dependence Graph (SDG) for VHDL introduced in [41] where the additional dependencies are represented as *flow edges*. However, the data flow information we store is simpler and mainly targeted towards the coverage estimation algorithm described later, in Section 3.5.

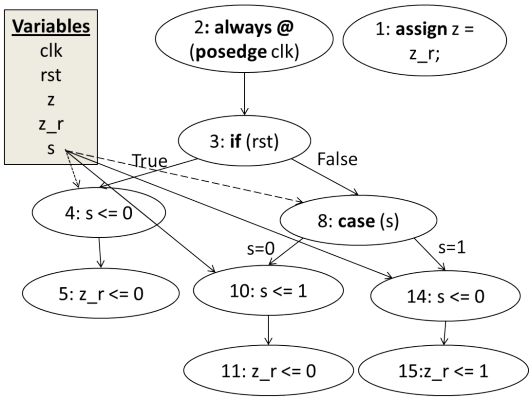
We store a list of variables used in the RTL code. The variables are classified into *inputs, outputs, internals* and *parameters*. The dependence information for a variable v is recorded in the form of *initial assignment*, *decisions* and *assignments* as described below.

- *Initial assignment:* This is only valid if v is an *internal* or an *output* variable of Verilog `reg` type. It points to the CFG node where the variable is assigned its initial value. This is typically the value when the reset signal is asserted, or in general, any value fully defined by inputs.
- *Decisions:* These are decision nodes which use v in the corresponding condition in the RTL.
- *Assignments:* These are assignment nodes with assignments to v, or in general, all assignments with v in the left hand side.

Example 1 Figure 3.1 illustrates the terms defined above with the help of an example Verilog RTL code. The CFG for the module consists of the CFGs for the two processes, viz. the continuous assignment on line 1 and the always process starting at line 2.

```
1. assign z = z_r;
2. always @ (posedge clk)
3.    if (rst) begin
4.       s <= 0;
5.       z_r <= 0;
6.    end
7.    else
8.       case (s)
9.          0: begin
10.              s <= 1;
11.              z_r <= 0;
12.           end
13.         1: begin
14.              s <= 0;
15.              z_r <= 1;
16.           end
17.      endcase
```

(a)



(b)

Figure 3.1: (a) Example Verilog RTL code for a design with inputs `clk, rst`, output `z` and internal variables `s, z_r`. (b) Control flow graph for the example RTL design augmented with variable dependency information in the form of *assignments* (solid lines) and *decisions* (dashed line) and *initial assignment* (dotted line) for variables. For clarity, nodes containing only keywords such as `begin` and `end` are not shown. Also, variable dependency information is shown only for the internal variable `s`.

It can be seen that each statement in the RTL code is mapped to a node in the CFG. Nodes corresponding to `if (rst)` and `case (s)` are decision nodes. Each of these has corresponding `end` nodes which are not shown for clarity.

The data flow information in the form of variable dependencies is also shown in Figure 3.1. Again for clarity, only the information for internal variable `s` is shown. The node corresponding to statement `s <= 0` on line 4 forms *initial assignment* for variable `s` (dotted line). Variable `s` is used in the decision corresponding to `case (s)` (dashed line) and assignments on lines 10 and 14 (solid lines).

3.3 Defining Code Coverage Metrics for Assertions

In this section, we define the two code coverage metrics for an assertion. We start by introducing some relevant terms.

We consider assertions of the form $ant \Rightarrow con$, where the antecedent ant is a conjunction of propositions and con is a single proposition. A proposition here is of the form $v == val$ where v is a variable in the RTL code. We also consider *temporal* assertions, which are assertions spanning multiple clock cycles. Assertions are represented in Linear Temporal Logic (LTL) [86] format.

For example, following is an assertion for the design in Example 1.

$$\neg rst \wedge s \Rightarrow X(z)$$

In words, this assertion expresses the property: 'if $rst == 0$ and $s == 1$ then in the next cycle, z should evaluate to 1'.

An assertion is said to *trigger* when its antecedent evaluates to true.

We call the set of statements reported as covered by the assertion 'a' according to our definition the *result set* of a. Therefore the proposed coverage metrics provide a notion of *statement coverage* of an assertion.

3.3.1 Simulation Based Code Coverage

We define the simulation based code coverage of an assertion such that it closely approximates the simulator statement coverage of a test that triggers the assertion. Our coverage definition as well as coverage analysis ensures that the coverage reported by our technique is an under-approximation or a conservative estimate of the simulator coverage. In other words, if a statement is reported as covered by our technique, it must be reported as covered by the simulator as well, while the converse may or may not be true.

Definition 1 *(Simulation based coverage) A statement s is covered by an assertion a if it belongs to one of the following categories.*

(1) s must be executed in order for the antecedent of a to evaluate to true.

(2) s is executed because the antecedent of a evaluates to true.

(3) s is executed irrespective of the evaluation of the antecedent of a.

(4) s does not belong to the above three categories, but execution of a statement from (1) or (2) ensures the execution of s.

We call the set of statements falling in categories (1) and (2) as the *backward cone* and the *forward cone* of the assertion, respectively. We use the term *ancillary cone* to denote the statements belonging to categories (3) and (4).

Note that (3) includes continuous assignments as well as unconditionally executed statements in each `always` process. As an example of a statement in

category (4), consider the following code snippet.

```
if (cond) begin
    s1;
    s2;
end
```

Suppose that `s1` belongs to the backward cone of a decision node, but not `s2`. However during simulation, `s2` must be executed if `s1` is executed. We include `s2` in the ancillary cone of the decision node.

Note that we include only those statements which are definitely executed, whenever a given assertion is triggered. As a result, if a test triggers a, the result set of a is a subset of the set of lines covered by the test. If multiple tests trigger the same assertion a, the result set is an intersection of the sets of lines covered by each of the tests.

Since this definition depends only on *triggering* of an assertion, the consequent as well as the correctness of the assertion is irrelevant to the coverage.

For the RTL design from Example 1, consider the following assertion:

$$a : \neg rst \wedge s \Rightarrow X(z).$$

The assertion a triggers when the antecedent $\neg rst \wedge s$ evaluates to true.

It can be concluded from Figure 3.1 that statements on lines 3, 8, 9 and 10 must be executed prior to variable `s` getting the value 1. These belong to the backward cone of a.

Statements on lines 3, 8, 13, 14 and 15 are executed due to the triggering of a, i.e. due to the variables rst and s evaluating to 0 and 1, respectively. Therefore, these statements are included in the forward cone of a.

Continuous assignments such as `1: assign z = z_r;` are executed irrespective of the triggering of a and therefore belong to category (3) from Definition 1. The statement `11: z_r <= 0;` is executed whenever line 10 is executed and therefore belongs to category (4) from Definition 1. These statements constitute the ancillary cone of a.

Figure 3.2 shows the CFG nodes covered by a in this case.

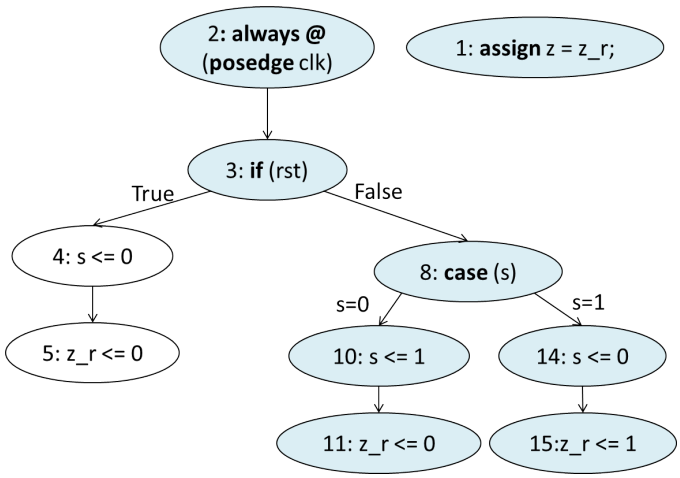


Figure 3.2: CFG nodes covered by a according to the simulation based coverage definition.

3.3.2 Correctness Based Code Coverage

In the context of formal verification, we define the coverage of an assertion based on its correctness. Essentially, a statement is said to be covered if its correct execution is necessary for the correctness of the assertion.

Definition 2 *(Correctness based coverage) A statement s in the RTL source code is said to be covered by assertion a if for some error or mutation injected at s, a fails formal verification in the mutated design.*

In this context, an *error* or *mutation* represents a logical bug such as an incorrect value assigned to a variable.

Since this definition depends on the correctness of the assertion, the number of cycles spanned by the assertion as well as the consequent are relevant to the coverage.

In [18], a mutation based definition of code coverage of an assertion is presented. However, the mutation considered involves removing the statement, and coverage is computed by checking if the assertion becomes vacuous in the mutant design. We consider logical bugs as mutations and compute coverage through analysis of the CFG for the RTL source code as described in Section 3.5.

Consider again the assertion a above for the code in Example 1. Statements on lines 1, 3, 8, 13 and 15 are included in the result set of a according to the correctness based definition.

Figure 3.3 shows the CFG nodes covered by a according to this definition.

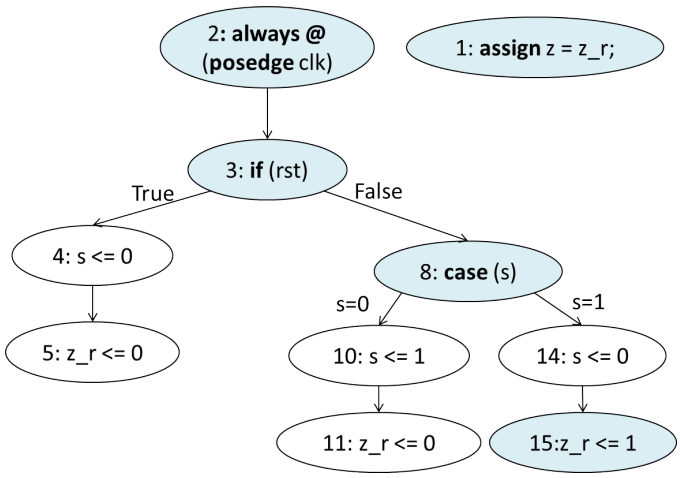


Figure 3.3: CFG nodes covered by a according to the correctness based coverage definition.

Given that the antecedent $\neg rst \wedge s$ holds, an error in one of these statements can make the consequent false and therefore make the assertion fail. For example, if the antecedent holds and either `z_r` or `z` is not assigned to the correct value, it will make the assertion fail.

A comparison between the result sets of a according to the two definitions show that statements on lines 9, 10 and 14 are included in simulation based coverage but not in correctness based coverage. Lines 9 and 10 fall in this category because they are executed prior to the antecedent becoming true and hence are irrelevant to the correctness of a. Line 14 is also unable to affect the correctness of a, and therefore it is not included in correctness based coverage.

3.4 Simulation Based Code Coverage Computation

We now describe our algorithm to compute the coverage of an assertion according to the simulation based coverage metric. Let a be an assertion for an RTL module M with CFG G. Let B_a, F_a and A_a be the sets of statements in backward, forward and ancillary cones of a, respectively. Let S_a be the result set of a. Then $S_a = B_a \cup F_a \cup A_a$.

3.4.1 Backward Cone

In order to find the statements in the backward cone of a, we look at each proposition in a individually. We find the statements that *must* be executed for the proposition to evaluate to true. Since a is a conjunction of such propositions, each of these propositions must evaluate to true for a to trigger. Therefore, if a statement must be executed for any of these propositions to evaluate to true, it must be executed for a to trigger. As a result, the union of backward cones of propositions gives the backward cone of a. We only consider propositions involving internal variables or outputs in this analysis.

Procedure 1 describes how we obtain the backward cone of each proposition. Consider a proposition $p : (v == val)$. We first find CFG nodes which assign the value val to variable v in $GetMatchingAssignments()$ (Procedure 2). Apart from direct assignment $v <= val$, we also look for indirect assignments, that is, assignments of the form $v <= v'$ such that v' in turn is assigned val directly or indirectly. If val happens to be the initial value of v, the CFG node corresponding to the initial assignment is immediately returned without further processing.

It is possible that v can be assigned val in multiple statements of a process. Since our goal is to find the statements that *must* be executed for a to trigger, we stop the backward traversal in such cases. Thus, we do not consider all possible execution paths that lead to v being assigned the value val.

When there is exactly one matching assignment n, it is included in B. If it is an indirect assignment, e.g. of the form $v <= v'$, a recursive call is made for a new proposition $p' : (v' == val)$.

Finally, all decision nodes that lie on the path from top level CFG node to the node corresponding to n are also included in B.

3.4.2 Forward Cone

To compute F_a, we use the information provided by a in the form of propositions. Since we assume that a has triggered, we have available a list of values L for some of the variables.

We look at all `always` processes containing a decision involving some $v \in L$. For each such process, we traverse the CFG starting from the top level node. When a decision node is encountered, we attempt to evaluate the corresponding conditional expression using the values in L. If it can be evaluated, we take the re-

Procedure 1 Extract backward cone of proposition $p : (v == val)$ in RTL module M

GetBackConeProposition(G, p)

Input: Extended CFG G of M, proposition $p : (v == val)$
Output: Set of statements B in backward cone of proposition p

$B \leftarrow \phi$
$N \leftarrow GetMatchingAssignments(p)$ {Find assignments that make p true}
if $\|N\| = 1$ **then**
  {Continue if exactly one matching assignment exists}
  Consider $n : (v <= rhs) \in N$
  $B \leftarrow B \cup n$
  if $\neg IsConstant(rhs)$ **then**
    {Recursive call in case rhs is also a variable}
    Let $p' \leftarrow (rhs == val)$
    $B \leftarrow B \cup GetBackConeProposition(G, p')$
  end if
  while $n \neq NULL$ **do**
    {Include all decision nodes on which n depends}
    if $IsDecisionNode(n)$ **then**
      $B \leftarrow B \cup n$
    end if
    $n \leftarrow Parent(n)$
  end while
end if

Procedure 2 Get CFG nodes with assignments matching proposition p

GetMatchingAssignments(G, p)

Input: Extended CFG G of M, proposition $p : (v == val)$
Output: Set of CFG N nodes containing assignments matching p

$N \leftarrow \phi$
if $val = v_{init}$ **then**
  $N \leftarrow \{Initial(v)\}$
else
  for all $n : (v <= rhs) \in Assignments(v)$ **do**
    if $rhs == val$ **then**
      $N \in N \cup \{n\}$
    else
      Let $p' \leftarrow (rhs == val)$
      Let $N' \leftarrow GetMatchingAssignments(G, p')$
      if $\{N'\} > 0$ **then**
        $N \leftarrow N \cup \{n\}$
      end if
    end if
  end for
end if

spective traverse along the respective edge. If the expression cannot be evaluated, we stop the traversal. The statements mapped to the nodes encountered during this traversal are included in F_a.

When the conditional expression for a cannot be represented as a conjunction of propositions, we simply start at a and traverse downwards until another decision node is encountered. The statements corresponding to the visited nodes are included in F_a.

3.4.3 Ancillary Cone

Extracting ancillary cone A_a consists of the following two steps, one for each of the categories (3) and (4) from Definition 1.

i. Firstly, all continuous assignments are included in A_a. Then, starting at the top level CFG node of each process, we traverse the CFG until a decision node is encountered. Statements corresponding to all visited nodes are included in A_a, since those are always executed irrespective of triggering of a.

ii. For each assignment node in B_a, we first traverse the CFG upwards until a decision node is reached. Statements corresponding to the visited nodes are included in A_a, since those lines are executed whenever the original assignment corresponding is executed. The same procedure is then repeated for downward traversal starting from the assignment node.

3.5 Correctness Based Code Coverage Computation

This section describes our algorithm to extract the result set of an assertion according to the correctness based coverage definition. It consists of two phases, viz. the *execution* phase and *coverage extraction* phase.

Consider an assertion $a : ant \Rightarrow con$ for module M, that spans k cycles. Let V be the set of variables in M. For a k cycle assertion, con is of the form $XX \ldots X(ktimes)(v_{out})$ or $XX \ldots X(ktimes)(\neg v_{out})$, where $v_{out} \in V$. Our goal is to find C_a, the result set of a according to the correctness based coverage definition. Procedure 3 describes how we compute C_a.

Procedure 3 Extract correctness based coverage of assertion $a : ant \Rightarrow con$ in RTL module M

GetFormalCoverage (G, a)

Input: Extended CFG for M (G), assertion a

Output: Set of lines C_a covered by a according to correctness based coverage definition

1: $B_{top} \leftarrow ConstructValueTable(G, a)$ {Execution phase: Construct a tree of value tables for n cycles and record triggers}
2: $C_a \leftarrow ExtractCoverage(G, B_{top})$ {Coverage extraction phase: Find coverage of a using value tables and triggers}

3.5.1 Execution Phase

In this phase, we execute the RTL for k cycles starting with the information in a (Procedure 4). In this process, we record the following information:

- We construct $\|V\| \times k$ tables of values of variables in k cycles. In particular the entry $B[v, i]$ in a value table B contains the value of variable v in cycle i. In other words, a value table is a table of propositions (variable-value pairs) in each of the k cycles. Multiple value tables correspond to different possible values of conditions corresponding to the decision nodes encountered during execution that cannot be evaluated to true or false.

- Apart from value tables, we also record triggers for each decision or assignment node that is visited. Triggers for a decision node are assignment nodes that make the decision true. Triggers for an assignment node include other assignment nodes which affect it and decisions nodes on which it depends.

When an assignment node $n : (v <= rhs)$ is encountered during execution, the value table B is updated with the value rhs if it is a constant. If it is not a constant, assignments to rhs encountered thus far are added to the set of triggers of n. The decision nodes on which this assignment depends are also added to the set of triggers of n. Essentially, these are the decision nodes that lie on the path in the CFG from n to a top-level node.

When a decision node n is encountered, we attempt to evaluate the condition in n using the values available in B. If it evaluates to true, we update its triggers and take the true branch in the CFG. If it evaluates to false, we take the false branch without updating the triggers. If the decision is unknown due to a lack of sufficient data in the value table, we split B into B_{left} and B_{right} corresponding to the true and false evaluations of n, respectively. As a result, at the end of the execution phase, we obtain a tree of value tables rooted at B_{top}.

Procedure 4 Construct a tree of value tables rooted at B_{top} using assertion a and record triggers

ConstructValueTable (G, a)

Input: Extended CFG for M (G), assertion a

Output: Tree of value tables rooted at B_{top}, triggers recorded in G

```
1: Initialize(B_top, a) {Initialize root table of values (B_top) using a}
2: for cycle = 1 → k do
3:    for all Leaf tables B in tree rooted at B_top do
4:       for all Processes P in M do
5:          n ← top(P)
6:          while n ≠ NULL do
7:             if IsAssignmentNode(n) then
8:                Let n : (v <= rhs)
9:                if IsConstant(rhs) then
10:                  {Update the value table}
11:                  B[v, cycle] ← rhs
12:               end if
13:               UpdateTriggers(n)
14:               n ← left(n)
15:            else
16:               {Decision node}
17:               if EvaluateDecision(n, B) = true then
18:                  UpdateTriggers(n)
19:                  n ← left(n)
20:               else if EvaluateDecision(n, B) = false then
21:                  n ← right(n)
22:               else
23:                  {Unknown decision}
24:                  (B_left, B_right) ← split(B)
25:               end if
26:            end if
27:         end while
28:      end for
29:   end for
30: end for
```

3.5.2 Coverage Extraction Phase

In the coverage extraction phase (Procedure 5), we look at each leaf B of the tree of value tables in turn. If v_{out} is assigned in cycle k in B and its value agrees with con, we have found a possible execution starting from ant that makes con true. We then include RTL lines for all nodes visited during this execution in C_a. We use the triggers recorded during the execution phase to traverse backwards recursively and obtain such nodes. This is implemented by the $BackwardTraversal()$ procedure on line 7. Pseudocode for that procedure is not shown due to space constraints.

Procedure 5 Extract correctness based coverage from value table tree and triggers

ExtractCoverage (G, B_{top}, a)

Input: Extended CFG for M (G), tree of value tables rooted at B_{top}, assertion $a : ant \Rightarrow con$

Output: Set of lines C_a covered by a according to correctness based coverage definition

```
 1: C_a ← φ
 2: Let p : (v_out, val) ← con
 3: for all Leaf tables B in tree rooted at B_top do
 4:     if B[v_out, k] = val then
 5:         {Value of v_out exists and matches the consequent of a}
 6:         for all Triggers t of p do
 7:             C_a ← C_a ∪ BackwardTraversal(t)
 8:         end for
 9:     end if
10: end for
```

Our correctness based coverage computation technique is guaranteed to find all statements such that erroneous execution of these statements causes the assertion to fail during formal verification. In some cases, our technique can report a statement as covered, but no mutation to that statement can cause the assertion to fail formal verification. This can only happen in the following two situations:

1. The mutation causes the assertion to be *vacuously true*. In this case, due to the injected mutation, the antecedent of the assertion never evaluates to true.

2. The mutation is *masked* by the logic and fails to affect the consequent and hence the correctness of the assertion.

In these situations, our technique computes a superset of the target result set of the assertion.

Table 3.1: Number of lines of RTL code, number of RTL statements and the number of assertions written for each module under consideration.

Module	Lines	Statements	Assertions
usbf_pa	257	134	4
usbf_pd	277	135	4
usbf_pe	620	302	5
usbf_idma	301	158	4
usbf_wb	137	53	4
or1200_dc_fsm	176	99	3
or1200_except	357	151	3

3.6 Experimental Results

In this section, we evaluate our correctness based coverage computation technique on a USB 2.0 protocol design and the OpenRISC processor design both from [43]. We demonstrate the simulation based correctness computation technique on an industrial design in Chapter 5, where we apply it to solve the problem of coverage extraction from emulation platforms.

In the USB design, we consider the packet assembler (`usbf_pa`), the packet disassembler (`usbf_pd`), the protocol engine (`usbf_pe`) and the internal DMA engine (`usbf_idma`) modules which form the protocol layer along with the wishbone interface module (`usbf_wb`). We consider the data cache controller (`or1200_dc_fsm`) and the exception unit (`or1200_except`) from the OpenRISC design. All experiments were performed on an Intel Core 2 Quad with 16GB RAM. Cadence Incisive Formal Verifier was used as the formal verifier.

Table 3.1 shows the number of lines of code and the number of RTL statements relevant for code coverage for each module considered. The statements include assignments and `if` and `case` conditional statements. For each of the seven modules, we manually wrote 3 to 5 assertions spanning up to 7 cycles for a total of 27 assertions. All assertions were formally verified against the corresponding design modules.

For each assertion, we extracted the set of statements covered according to the correctness based definition. Covered statements were mutated and the assertion was run through the formal verifier again along with the mutated designs. Following mutations were injected for each type of statement:

`if` conditions: The condition was negated, or in other words, the *true* and *false* branches were flipped.

`case` conditions: Code block corresponding to one case was interchanged with that for another case.

Assignments: Right hand side of the assignment was changed, e.g. `b = 0` was changed to `b = 1`.

Table 3.2 shows detailed results for all assertions considered in this experiment. Time and memory usage for coverage computation of every assertion is shown. We also give the maximum depth of the value table tree constructed during the execution phase (Section 3.5.1). Finally, we show the number of statements covered and number of statements such that a mutation injected at the statement is detected, or in other words, the assertion fails formal verification with the mutated design.

Recall that the value table tree depth is determined by the number of unknown decisions encountered during the execution phase. The memory usage of coverage computation can be attributed mainly to the CFG and the value table tree. As the value table tree depth increases, the value table tree contributes towards most of the memory usage. The current implementation of coverage computation is such that the size of the value table tree is exponential in its depth. As a result, we can see that for a given module, the total memory usage also increases exponentially with the value table tree depth. For example, in the case of `usbf_pa`, the memory usage increases from 361 kB for value table tree depth of 1 (a_3) to 1.24 GB for value table tree depth of 14 (a_4). In the future, we aim to make the value table tree implementation more efficient and avoid this exponential memory cost.

The memory usage of coverage computation for a module averaged over all assertions is found to increase with the size of the module. This can be attributed to the larger CFG and value tables for larger modules with higher numbers of variables.

Runtime of coverage computation also behaves in a way similar to memory usage in that it increases with the size of the module and for a given module, with the value table tree depth.

It can be seen that on average, only about 8% of total statements were reported as covered by each assertion, which shows that our technique can effectively find the statements truly relevant to the correctness of an assertion.

For every assertion, mutations at almost all covered statements were detected. On careful analysis of the source code with the assertion, we find that the mutations are not detected in only the situations mentioned in Section 3.5.

Table 3.2: Results for each of the 27 assertions. Column three gives the length in number of cycles of each assertion while column four gives the maximum depth of the value table tree during coverage computation for that assertion. The time and memory usage during coverage computation and the number of statements covered are shown in the next three columns. Number of statements such that the mutation injected in the statement was detected is shown in the last column.

Module	Assertion	Length	Tree depth	Time (s)	Memory	Covered state-ments	Mutations detected
usbf_pa	a_1	1	13	0.18	83 MB	5	5
	a_2	2	2	0.01	416 kB	5	4
	a_3	3	1	0.01	361 kB	6	6
	a_4	4	14	2.78	1.24 GB	19	15
usbf_pd	a_5	1	3	0.01	589 kB	4	4
	a_6	2	5	0.02	1.49 MB	16	13
	a_7	3	1	0.02	625 kB	12	11
	a_8	4	10	0.11	53.3 MB	7	7
usbf_pe	a_9	2	5	0.05	8.4 MB	8	8
	a_{10}	2	4	0.03	1.64 MB	5	5
	a_{11}	3	5	0.05	16.1 MB	4	4
	a_{12}	4	7	0.15	66.3 MB	16	14
	a_{13}	7	18	8.09	3.2 GB	25	25
usbf_idma	a_{14}	1	17	0.52	251 MB	4	4
	a_{15}	2	13	0.05	19.5 MB	26	25
	a_{16}	3	15	0.2	89 MB	8	8
	a_{17}	4	5	0.01	3.46 MB	15	15
usbf_wb	a_{18}	1	2	$<$0.01	252 kB	5	5
	a_{19}	2	0	0.01	177 kB	7	7
	a_{20}	4	3	0.01	725 kB	5	5
	a_{21}	6	6	0.02	3.69 MB	18	17
or1200 _dc_fsm	a_{22}	2	8	0.01	904 kB	7	3
	a_{23}	3	5	0.01	1.6 MB	15	14
	a_{24}	4	1	0.01	472 kB	14	10
or1200 _except	a_{25}	2	9	0.02	4.57 MB	5	5
	a_{26}	3	18	0.03	15.8 MB	8	3
	a_{27}	4	4	0.02	1.87 MB	8	8

We now describe the results for assertion a_2 for the USB packet assembler module in further detail.

$$a_2\text{: } \texttt{rst} \wedge \texttt{tx\_ready} \wedge (\texttt{state = CRC2}) \Rightarrow X\,(\texttt{state = IDLE})$$

Figure 3.4 shows the relevant part of the RTL code for the module, where the statements covered by a_2 are underlined. The lines of code shown constitute a state machine with five states: `IDLE, WAIT, DATA, CRC1, CRC2`.

```
1. always @(posedge clk)
2.    if(!rst)  state <= IDLE;
3.    else      state <= next_state;

4. always @(state or send_data or tx_ready or tx_valid_r or send_zero_length_r)
5.    begin
6.       //assignments
7.       case(state)
8.       //other cases: IDLE, DATA, WAIT, CRC1
9.         CRC2:
10.        begin
11.             //assignments
12.             if(tx_ready)
13.                begin
14.                     next_state = IDLE;
15.                end
16.             else
17.                begin
18.                     last = 1'b1;
19.                end
20.         end
21.       endcase
```

Figure 3.4: RTL code snippet from the USB packet assembler module relevant to assertion a_2: $\texttt{rst} \wedge \texttt{tx\_ready} \wedge (\texttt{state = CRC2}) \Rightarrow X\,(\texttt{state = IDLE})$. Statements reported as covered by a_2 by our technique are underlined.

Each covered statement was mutated one at a time, as shown in Table 3.3.

We found that a_2 failed formal verification in mutated designs with mutations in lines 9, 12 and 14. For instance, consider the mutation to the assignment in line 14. In this case `next_state` is assigned an the wrong value `DATA` instead of the correct value `IDLE`. The erroneous value propagates to the variable `state` in the next cycle which causes a_2 to be false, since the consequent of a_2 checks if the `state` equals `IDLE` in the next cycle.

Mutation in the statement `state <= next_state` (line 3) did not cause a_2 to fail formal verification. Due to the mutation, the only values `state` can take are: `IDLE` and `DATA`. Since the antecedent of a_2 requires `state = CRC2`, it never evaluates to true and a_2 passes the formal verification vacuously.

Table 3.3: Mutations injected in the statements covered by assertion a_2 for the USB packet assembler module. In the case of the mutation in line nine, the statement `IDLE:` (not shown in Figure 3.4) was changed to `CRC2:` in turn. Thus the code blocks corresponding to the `IDLE` and `CRC2` cases were interchanged.

Line number	Original statement	Mutated statement
2	`if(!rst)`	`if(rst)`
3	`state <= next_state`	`state <= DATA`
9	`CRC2:`	`IDLE:`
12	`if(tx_ready)`	`if(!tx_ready)`
14	`next_state = IDLE`	`next_state = DATA`

Mutation in `if(!rst)` (line 2) was also undetected. In this case, `next_state` had the value `IDLE` which is also the value of `state` on reset. Therefore `state` was assigned value `IDLE` in both branches of the `if` statement. In other words, the mutation was *masked* by the logic in the design and hence had no effect on the correctness of a_2.

CHAPTER 4

FAULT COVERAGE ANALYSIS OF ASSERTIONS

4.1 Introduction

In this chapter, we describe our approach of fault coverage analysis and fault criticality estimation. We consider the assertions automatically generated by the GoldMine tool [40, 44]. The fault model considered for this analysis is the Single Event Upset (SEU), which is typically used to model soft errors in hardware designs [87].

4.1.1 GoldMine Assertion Generation Tool

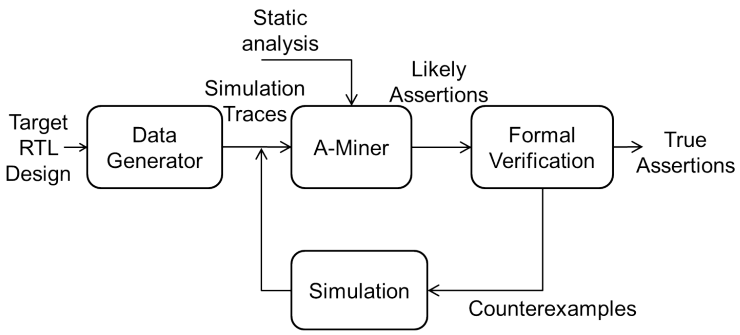


Figure 4.1: GoldMine tool flow from [44].

In the GoldMine tool flow (Figure 4.1), a target RTL design is simulated for a fixed number of cycles using random input patterns. The simulation traces generated are used as data by the data mining stage (A-Miner) comprising a decision tree based supervised learning algorithm. Static analysis of the RTL design is carried out in order to extract the set of variables which affect a design output. This set is known as the *logic cone* of the output. A-miner is restricted to analyzing the data only for the logic cone of an output. GoldMine can generate *temporal*

assertions or sequential assertions, which are relevant for detecting SEUs. A *mining window length* given to A-Miner provides the number of cycles for which the design should be unrolled to capture sequential behavior. A-Miner guesses the likely assertions in the design.

Formal verification is used to extract true assertions from the set of likely assertions. The true assertions in an iteration are retained. The assertions that fail the formal verification phase are poor guesses by the data miner. The formal verifier produces counterexample traces for every failed assertion. These traces are appended into the simulation testbench in a following iteration of GoldMine. This process is repeated until no assertions fail. The final set of true GoldMine assertions is guaranteed to capture the complete functionality of a design output [44].

4.1.2 Fault Coverage and Criticality Evaluation

Since GoldMine automatically generates a large number of assertions for a design, it is important to evaluate the quality of generated assertions. This evaluation can be used to rank the assertions and only use the highly ranked assertions through the various stages of verification. In [40], subjective ranking by designers is used for this purpose. However, a more objective metric which can be automatically computed is more valuable. To that end, in this work we evaluate the assertions in terms of the number of faults they detect, using the SEU fault model.

Both our fault injection and fault coverage evaluation follow a formal approach similar to [71]. We inject faults modeled as single event upsets (SEUs) into the state variables (flip-flops/latches) in RTL, thereby creating a *faulty design*. We use GoldMine to generate assertions for the *fault-free design*. The faulty design is run with these assertions through a formal verifier. If an assertion that is true in the fault-free design fails in a faulty design, it is said to detect the injected fault. The faults covered by assertions in one iteration are removed from the set of uncovered faults. This process is repeated until all faults are detected or until no more counterexamples are generated by GoldMine.

The fault coverage analysis can also be turned around to estimate the *criticality* of a fault. Assertions that detect a fault cover different paths in the design through which the fault propagates to the output. Therefore, the number of GoldMine assertions that detect a particular fault can provide an estimate of the criticality or importance of that fault for overall design vulnerability to soft errors. These

estimates can also be used for selective protection of flip-flops/latches [71,73,75].

In this work, we formally show how GoldMine assertions detect the injected SEUs, by representing the fault-free and faulty designs as a finite state machine (FSM). We demonstrate our technique on the SpaceWire [88] communication controller design from OpenCores [43].

4.2 Theoretical Framework

In this section, we describe how an RTL design can be represented as an FSM. We then show how an injected fault changes the FSM.

4.2.1 RTL Design as an FSM

We use M to denote both the RTL design as well as the corresponding FSM. Thus $M = \langle I, Z, V, s_i, \delta, \rho \rangle$, where I is the set of inputs, Z is the set of outputs, and V is the set of state variables which induce the state space $S = 2^V$. Let s_i be the initial state of M. The transition function $\delta : 2^V \times 2^I \rightarrow 2^V \setminus \varnothing$ maps every state and input assignment to a successor state, and $\rho : 2^V \rightarrow 2^Z$ is an output function that maps each state to the set of outputs that are true in it. It is required that $I \cap V = I \cap Z = \varnothing$; however, it is possible that $Z \cap V \neq \varnothing$.

Given a k-cycle input sequence $\xi_k = i_0, i_1, \ldots, i_{k-1} \in (2^I)^k$ and a starting state $s_0 \in S$, M traverses a fixed sequence of states $\wp = s_1, s_2, \ldots, s_k \in S^k$ such that $\delta(s_j, i_j) = s_{j+1}\ \forall j = 1, 2, \ldots, k-1$.

The clock and reset input signals are not included in I. We further assume that the reset input is synchronous and whenever it is asserted, M transitions to its initial state s_i. It should be noted that I, V and Z contain only single-bit signals. For a bit-vector in the RTL design, we add a signal for every bit of the vector.

The logic cone of a design output z is the set of all variables in the design that affect z. We use $V_z \subseteq V$ to denote the state variables in the logic cone of z.

Figure 4.2(a) shows an example Verilog RTL design. The corresponding FSM is shown in Figure 4.2(b). Here, $I = \{a\}, V = \{v, z\}$ and $Z = \{z\}$. We show the value of input a on the edges, value of state variable v at the top of each state and value of output z at the bottom of each state. Here we denote each state by a vector formed by concatenating values of v and x. For example, if M is in the state 00 (i.e. $v = 0$ and $z = 0$) when $a = 1$ is received, it goes to the state 10.

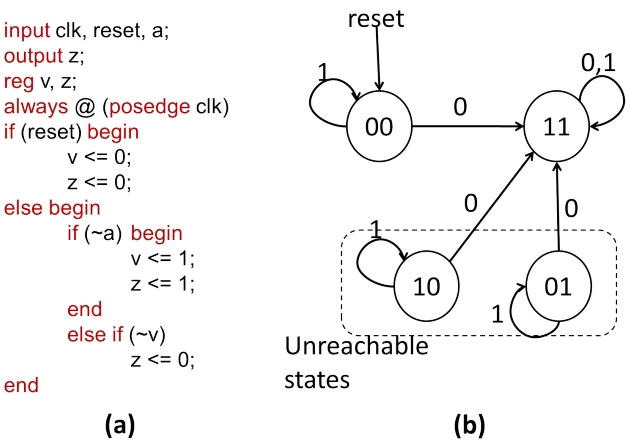


Figure 4.2: (a) A Verilog RTL design and (b) the corresponding finite state machine. In (b), we have shown the value of input a on the edges and the value of the vector $\{v, z\}$ inside each state. The bottom two states are not reachable from the initial state.

The initial state of M is s_i = 00. When the two-cycle input sequence $\xi_2 = 0, 1$ is applied to M in the state 00, M traverses the state sequence $\wp = 11, 11$.

It should be noted that the bottom two states in Figure 4.2(b) are unreachable from the initial state. The significance of unreachable states for fault detection is discussed later in Section 4.3.4.

4.2.2 Fault Model

We focus on soft errors in sequential elements. We consider only effective faults, i.e. the faults that are able to flip the state of the sequential element.

Soft errors at sequential elements can be modeled as *single event upsets (SEUs)* at corresponding state variables in the RTL design. An SEU at a variable corresponds to a bit-flip in the variable at an arbitrary cycle of operation. The design behaves normally during the remaining cycles.

We employ the single event upset fault model introduced in [71] to inject a fault into a Verilog RTL design. The original RTL code is instrumented to achieve the fault injection.

Consider a state variable $v \in V_z$ in the design where we wish to inject a fault. We define a new state variable SEU and a single-bit input $injectSEU$. $SEU = 0$ implies that the bit-flip at v has not yet occurred. In this state, if $injectSEU$ is 1 in a particular cycle, v is flipped and SEU changes to 1. This is achieved

by instrumenting assignments to v in the RTL code. In all subsequent cycles of operation, since the fault has already taken place, the input $injectSEU$ is simply ignored and the design behaves normally. As a result, only in the first cycle in which $injectSEU$ is 1, an SEU is injected at v. The variable SEU thus limits the fault to a single-event upset and prevents multiple bit-flips. Figure 4.3 models this behavior as a finite state machine M_f. From now on, we refer to SEU at v simply as a fault at v.

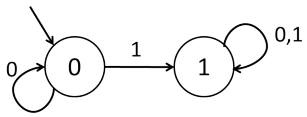


Figure 4.3: A finite state machine model for an SEU. The value of the input $injectSEU$ is shown on the edges and the value of the variable SEU is shown inside each state.

We use M_v to denote the instrumented or *faulty* RTL design as well as the corresponding FSM. $M_v = \langle I', Z', V', s'_i, \delta', \rho' \rangle$ is the *product machine* [89] obtained by combining M and M_f. Therefore, $I' = I \cup \{injectSEU\}, Z' = Z, V' = V \cup \{SEU\}$ and $s'_i = s_i \cup \{SEU = 0\}$. The state space of M_v is denoted by $S' = 2^{V'}$. For $s' \in S'$ with $SEU = 0$, we define s'_v to be the state which differs from s' only in the value of v and SEU. Values of all other state variables are equal in states s' and s'_v. Thus, s'_v is the *faulty state* corresponding to s'.

When a fault is injected at v in the RTL design in Figure 4.2, the faulty design formed is shown in Figure 4.4. We concatenate the values of inputs $injectSEU$ and a (in this order) to form an input vector which is shown on every edge in the faulty design. Similarly, the values of state variables SEU and v are concatenated and the resulting vector is shown at the top of each state. Since M_f has no outputs, the only output is z, which is shown at the bottom of each state. The bottom three states shaded blue have $SEU = 1$. Unreachable states in M_v are not shown since they do not play a role in fault detection.

The top two states have $SEU = 0$, which means that the bit-flip at v has not occurred. When M_v is in one of these states, its behavior is identical to M whenever $injectSEU = 0$. When $injectSEU = 1$, a bit-flip at v occurs and M_v transitions to one of the shaded states. For example, suppose the input vector $(injectSEU, a) = 11$ is received in state with $(SEU, v, z) = 000$. Since $a = 1$ and $v = 0$, the next state should also have $v = 0$ due to the self loop in Figure 4.2(b). However, $injectSEU = 1$ causes a bit-flip at v and the next state has

$v = 1$. Since the bit-flip has taken place, the SEU variable is set in the next state. Finally, the next state has $z = 0$ because assignments to z are not modified while constructing M_v.

Once M_v reaches one of the shaded states, the input $injectSEU$ becomes a 'don't care'. This is evident from Figure 4.4 which shows that the behavior in each of the three blue states is independent of the value of $injectSEU$. For example, if $a = 0$ is received when M_v is in the state with $(SEU, v, z) = 110$, the next state is 111, independent of the value of $injectSEU$.

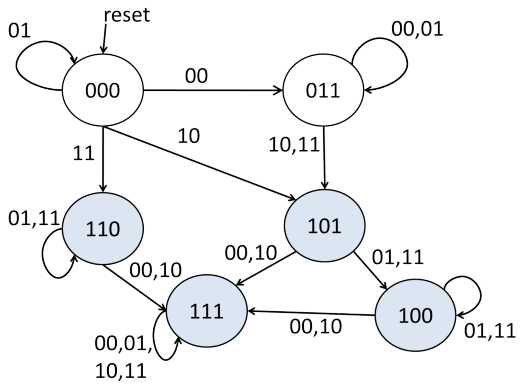


Figure 4.4: Faulty design obtained after injecting a fault at v in the design in Figure 4.2. Every transition is labeled with $\{injectSEU, a\}$ vector. Every state is labeled with $\{SEU, v, z\}$ vector. States with $SEU = 1$ are shown in blue. Unreachable states are not shown.

We use formal verification for evaluating fault coverage of a set of assertions. Specifically, the true GoldMine assertions generated from the fault-free design are checked against the faulty design. Formal verification exhaustively explores the states in the faulty design to test the validity of each assertion. Due to the injected fault, some of these assertions do not hold in the faulty design. These assertions detect the injected fault. Similarly to the coverage definition in [36], this can be formalized as follows.

Definition 3 *An assertion A detects (covers) a fault at a state variable $v \in V$ if $M \models A$ but $M_v \not\models A$.*

Definition 4 *Fault coverage of an assertion A is the fraction $\frac{|V_{detected}}{|V|}$ where $V_{detected} = \{v \in V | M \models A \wedge M_v \not\models A\}$.*

Note that according to Definition 3, an assertion detects a fault if its antecedent evaluates to true and consequent evaluates to false in the faulty design. In other

words, the assertion is said to detect (cover) a fault at a state variable if it detects the fault for some variable assignment that causes the antecedent to evaluate to true.

We inject faults at all state variables in the logic cone of z. Since these state variables correspond to sequential elements at circuit-level, our fault injection is exhaustive w.r.t. all sequential elements that affect z. Since our fault injection is based on a formal approach, we ensure that our fault injection is exhaustive w.r.t. the time of occurrence of the faults at the state variables under consideration.

4.3 GoldMine Assertions as Fault Detectors

We now give insights into how a GoldMine assertion is able to detect the injected faults. We first describe the format of temporal assertions generated by GoldMine. Assertions in this format correspond to a set of state sequences in the design. If one of these sequences is able to propagate a fault to the output, the assertion fails in the faulty design and hence detects the fault.

4.3.1 Format of GoldMine Assertions

The temporal assertions generated by GoldMine are of a specific format and correspond to a subset of all possible temporal assertions. In order to understand how GoldMine assertions detect faults, it is necessary to analyze how this format affects our fault injection and detection technique.

Let $A : ant \Rightarrow con$ be a GoldMine temporal assertion generated for output z which spans $k+1$ cycles. Then ant is k cycles long and is of the form:

$ant_0 \wedge X(ant_1) \wedge XX(ant_2) \wedge \ldots \wedge XX \ldots X(k-1 \text{ times})(ant_{k-1})$.

Each ant_i is a Boolean conjunction of propositions (variable-value pairs). Moreover, ant_0 contains propositions on $I \cup V$, but each ant_i for $i = 1, \ldots, k-1$ contains propositions on I. Thus the antecedent can contain propositions on the state variables only in its initial cycle. In all subsequent cycles, it contains propositions involving the inputs only. When the values of state variables in the initial cycle and values of inputs up to cycle $i-1$ are specified, the value of a state variable in cycle i is uniquely determined. As a result, the assertions can capture the complete functionality of z over the previous k cycles even in the presence of this

restriction. On the other hand, this restriction on ant helps in fault detection as explained later in Section 4.3.3.

The consequent con is of the form:

$XX \ldots X(k \text{ times})(z) = 0$ or

$XX \ldots X(k \text{ times})(z) = 1$.

In other words, the consequent always consists of a single proposition on the output. The decision tree-based algorithm used in the data mining stage of GoldMine constructs a separate decision tree for each output, and as a result, the consequents of the generated assertions satisfy this condition. It works to our advantage during fault detection since we only focus on the faults that can propagate to a given output. An assertion generated for z can therefore detect the effect of a fault on z, which is the desired functionality of a fault detector.

4.3.2 State Sequences Covered by Assertions

Due to the specific format of A as described above, ant corresponds to a set of starting states together with a set of k-cycle input sequences. Since ant_0 can contain a Boolean conjunction of propositions on the state variables, it corresponds to a set of states in M. If all state variables are present in ant_0, it corresponds to a unique state. If none are present, it corresponds to all the states. Similarly, each ant_i corresponds to a set of single cycle input patterns and together they constitute a set of k-cycle input sequences. Consequently, ant corresponds to or *covers* a set of state sequences (paths), one for every combination of starting states and input sequences.

Since the faulty design M_v is constructed as a product machine of M and M_f, all inputs, state variables and outputs of M are present in M_v. Therefore, ant corresponds to a set of state sequences in M_v as well.

For example, consider the assertion $A1 : (v \wedge a) \wedge X(a) \Rightarrow XX(z)$, generated for design M (Figure 4.2). Here $ant = (v \wedge a) \wedge X(a)$ and $con = XX(z)$. Figure 4.5(a) lists the starting state, input pattern and state sequence in M covered by ant. Only the reachable states are considered as starting states. Since ant_0 contains the proposition $v = 1$, 11 is the only possible starting state. Further as ant_0 and ant_1 both contain the proposition $a = 1$, the only possible input sequence is $1, 1$.

The state sequences covered by ant in the faulty design M_v (Figure 4.4) are

A1: (v ∧ a) ∧ X(a) => XX(z)

Starting state (*v, z*)	2-cycle input sequence (*a*)
	1, 1
11	11, 11

(a)

Starting state (*SEU, v, z*)	2-cycle input sequences (*injectSEU, a*)			
	01, 01	01, 11	11, 01	11, 11
011	011, 011	011, 101	101, 100	101, 100
110	110, 110	110, 110	110, 110	110, 110
111	111, 111	111, 111	111, 111	111, 111

(b)

Figure 4.5: State sequences covered by the the antecedent ant of $A1$ in (a) fault-free design M (Figure 4.2) and (b) faulty design M_v (Figure 4.4) for each combination of starting states and input sequences covered by ant. The number of state sequences in M_v is much larger than in M due to the additional signals $injectSEU$ and SEU.

listed in Figure 4.5(b). As ant_0 does not contain any proposition on the state variable SEU, it corresponds to three possible starting states, viz. $011, 110, 111$. Since neither of ant_0 and ant_1 contains a proposition on $injectSEU$, it can be either 0 or 1 in each cycle of the two-cycle input sequence. The sets of starting states and input sequences in turn lead to a set of 12 state sequences. It can be seen that due to the additional signals in M_v, the same ant covers a larger number of state sequences compared to M.

4.3.3 Fault Propagation to Output and Detection

Since we wish to use the assertions generated for outputs as fault detectors, the propagation of a fault to the outputs is necessary for its detection. Specifically, consider $s' \in S'$ with $SEU = 0$ and the corresponding faulty state s'_v (Section 4.2.2). Assume that s' is reachable from the initial state s'_i. Let ξ_r be an r-cycle input sequence. Let $s', s'_1, \ldots, s'_r$ and $s'_v, s'_{v1}, \ldots, s'_{vr}$ be the state sequences traversed when ξ_r is applied to M_v when it is in states s' and s'_v. Then if the values of z in s'_r and s'_{vr} are not equal, we can say that ξ_r propagates the fault at v to the output z. If no such s' and ξ_r exist, the fault cannot propagate to z in r cycles.

Let $A : ant \Rightarrow con$ be a true assertion in M such that ant is $k > r$ cycles long

and corresponds to a starting state $s_0 \in S'$ and a k-cycle input sequence ξ_k which satisfy the following conditions:

- $injectSEU = 0$ in each cycle of ξ_k.
- $\xi_k = \xi_{k-r}, \xi_r$, i.e. the last r elements of ξ_k form ξ_r, and ξ_{k-r} is the corresponding prefix sequence.
- When ξ_{k-r} is applied to M_v in state s_0, the resulting state is s'.

Suppose A is being checked against M_v in formal verification as described in Section 4.2.2. Due to ξ_k, the state sequence explored is $s_0, \ldots, s', s'_1, \ldots, s'_r$. The consequent con corresponds to the value of z in s'_r. Since A is generated from M, ant does not contain any proposition on $injectSEU$. Therefore, ant also corresponds to an input sequence which is identical to ξ_k except for the value of $injectSEU$ in cycle $k - r$. Due to this input sequence, the explored state sequence is $s_0, \ldots, s'_v, s'_{v1}, \ldots, s'_{vr}$. In this case, ant is true while con is false. Consequently, $M_v \not\models A$ and since $M \models A$, A detects the fault at v.

We illustrate the fault propagation and detection with an example. Let $s' = 011$ and the corresponding faulty state $s'_v = 101$, which differs from s' only in the values of SEU and v. When the single-cycle input sequence $\xi_1 = 01$ is applied to M_v in state s' and s'_v, values of z in the resulting state are 1 and 0, respectively. Therefore, ξ_1 propagates the fault at v to z.

From Figure 4.5(b) for $A1$, consider the starting state $s_0 = 011$ and the two-cycle input sequence $\xi_2 = 01, 01$. The last element of ξ_2 forms ξ_1 and the prefix sequence is also $\xi'_1 = 01$. When ξ_2 is applied to M_v in s_0, the state sequence followed is $011, 011, 011$. In the last state, $z = 1$, which agrees with the consequent of $A1$. If ξ_2 is modified by changing the value of $injectSEU$ in cycle 1, the resulting input sequence $11, 01$ gives rise to the state sequence $011, 101, 100$ (Figure 4.6). In the last state, $z = 0$. Therefore $A1$ is declared false in formal verification, thus detecting the fault.

We can now explain how the format of GoldMine temporal assertions helps in fault detection. As described in Section 4.3.1, the antecedent of a GoldMine assertion can contain propositions on state variables only in the first cycle. If this condition is relaxed, GoldMine can generate an assertion $A1'$ given by

$$A1' : (v \wedge a) \wedge X(v \wedge a) \Rightarrow XX(z)$$

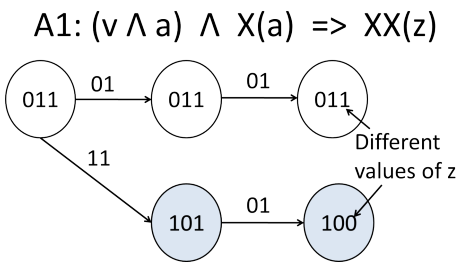


Figure 4.6: Fault at v detected by $A1$ for the design from Figure 4.4.

instead of $A1$ but which captures the same behavior in M as $A1$. Note that the state variable v is present in the second cycle of the antecedent of $A1'$. When the input sequence $11, 01$ is applied while $A1'$ is being checked against M_v, ant becomes false in the second cycle. As a result, $A1'$ is declared true and fails to detect the fault at v.

4.3.4 Fault Detection due to Newly Reachable States

We find that the presence of a fault makes some states that are unreachable in the original design reachable in the faulty design. These *newly reachable states* facilitate the detection of additional faults. We illustrate this phenomenon using the example design of Figure 4.2.

From the discussion in Section 4.3.3, for an assertion to detect a fault at v that propagates to output z in r cycles, the assertion must span at least $(r + 1)$ cycles. However, if an unreachable state in M becomes reachable in M_v, it is possible that a shorter assertion detects the fault.

The fault at v in the example design M of Figure 4.2 takes one cycle to propagate to z and therefore requires two-cycle assertions for detection.

If a mining window of one cycle is used while generating assertions, the following is one of the true assertions:

$$A2 : v \wedge a \Rightarrow X(z).$$

It can be observed from Figure 4.2 that $A2$ is not true starting from the state $\{v, z\}$ = 10. However, since this state is not reachable from the initial state, $A2$ is proved true in M.

In the faulty design M_v, the state $\{SEU, v, z\}$ = 110 is reachable from the initial state. In other words, the state with $v = 1, z = 0$ which was previously

unreachable has become reachable due to the fault. When the one-cycle input sequence $\xi = 1$ covered by $A2$ is applied to M_v in this state, $z = 0$ in the next cycle. Therefore, $A2$ is declared false in M_v, thus detecting the fault at v. Note that here the fault at v which propagates to z in one cycle is detected by $A2$ which spans only one cycle.

In general, some faults that propagate to the output in r cycles get detected by assertions generated from a mining window of r cycles. This helps in reducing the memory and time costs of the fault detector generation process.

4.4 Experimental Results

We demonstrate our fault coverage evaluation and fault criticality estimation approach on the control state machine of an end node of the SpaceWire [88] network. For GoldMine, a mining window of three cycles was used for the SpaceWire experiments. We generated assertions for four outputs of the module: RST_tx_o, err_sqc, enTx_o and enRx_o. Faults were injected at 25 state bits that are in the logic cone of all outputs under consideration. Cadence IFV was used as the formal verifier for fault experiments. Experiments were performed on an Intel Core 2 Quad with 16 GB RAM.

4.4.1 Fault Coverage Based Ranking

We ranked the assertions generated for SpaceWire FSM where assertions detecting ($\leq$10%), (10%-20%), (20%-30%) and (>30%) faults were assigned ranks 1, 2, 3 and 4, respectively. We then computed the percentage of assertions belonging to each rank (Figure 4.7). In this experiment, most of the assertions (82%) have rank 1 or 2, i.e. they detect up to 20% of injected faults. However, there are 4% assertions belonging to rank 4 and we have also found that about 6% of those detect more than 80% of the injected faults.

4.4.2 Fault Criticality

In Figure 4.8, we show the percentage of total assertions generated (D_z) for each output of SpaceWire FSM that detect a given fault. This number gives an estimate

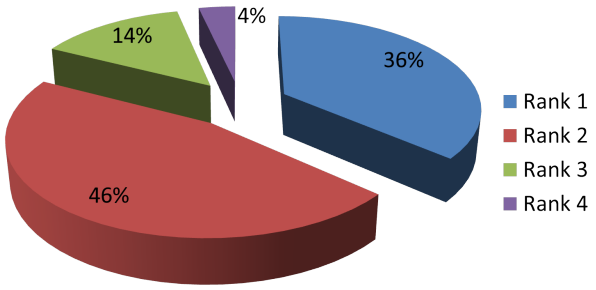


Figure 4.7: Percentage of total assertions belonging to each rank. Rank 1 : $\leq$10% faults detected, Rank 2: 10%-20% faults detected, Rank 3: 20%-30% faults detected and Rank 4: $>$30% faults detected.

of criticality or importance of the fault for overall design vulnerability. The faults considered are in the combined logic cone of all the outputs. For the variables `state[5:0]` and `timer[13:0]`, the number of assertions is averaged over individual bits.

We can see from the figure that a relatively high percentage of assertions (25% - 78%) detect faults at variable `state`. Therefore, faults at `state` would be most critical for all outputs considered. This is confirmed by the RTL description which shows a direct dependence of these outputs on `state`. The variable `state` is used as a switching variable of a `case` statement in the RTL. Since the rest of the variables can only affect the outputs by changing `state`, a lower percentage of assertions detect faults at those variables (up to 31%).

In [71], the authors have demonstrated that manually written assertions based on the SpaceWire specifications were not able to detect a fault at `HASgotBit`. GoldMine assertions, however, manage to detect this fault. At the same time, Figure 4.8 also illustrates that this fault is hard to detect, by showing a very low percentage of assertions detecting this fault (0.02% - 4%), thereby confirming the inference in [71].

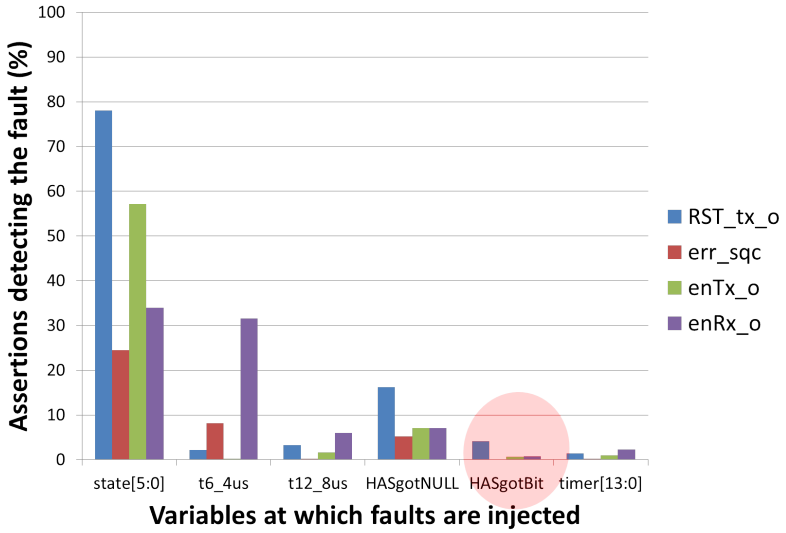


Figure 4.8: Percentage of generated assertions that detect a particular fault for SpaceWire control FSM. Fault at the highlighted variable could not be detected by a set of manually written assertions as reported in [71].

CHAPTER 5

CODE COVERAGE EXTRACTION FROM EMULATION AND PROTOTYPING PLATFORMS

5.1 Introduction

In this chapter, we define our work for code coverage extraction from emulation and prototyping platforms. We apply our simulation based coverage analysis technique from Section 3.4 to determine the mapping of conditional statements in the RTL code to the rest of the code. Simulation based coverage for an assertion (Definition 1) relates the *triggering* of an assertion to the execution of statements in the code, where triggering refers to the antecedent condition evaluating to true. We extend this definition to the conditions that are part of `if` and `case` statements in the code. Since conditions are encoded as decision nodes in the CFG framework (Section 3.2), we refer to the condition evaluating to true as the *triggering* of the corresponding decision node.

For each decision node, we identify the set of statements such that triggering of the decision node ensures execution of these statements. This set consists of the statements that must be executed for the decision node to trigger (*backward cone*) and the statements conditioned on the decision node (*forward cone*). This static analysis is performed in a way that the coverage from our technique closely approximates the statement coverage reported by a simulator.

We add a block of code to record the triggering of each decision node which gets synthesized into additional logic. During emulation, we use the added logic to identify the decision nodes that triggered during running of a test. This trigger information is cross referenced with the source code to compute code coverage statistics.

The total area cost of the added logic is roughly proportional to the number of decision nodes. In order to keep the area overheads under control, we optimize the set of decision nodes according to user specified constraints. These "knobs" allow the user to make the best use of the resources available on the emulation

platform.

We demonstrate our technique through experiments on an industrial design emulated on a Xilinx FPGA platform. We use a signal processing module that generates timing recovery strobes for a receiver front end. The runtime of our technique was within 30 seconds for the design with a few thousand lines of source code, which shows that the technique is scalable and efficient. The code coverage reported using our technique is compared to that from an RTL simulator. We find that with around 10% FPGA usage overhead, the error in reported coverage by our technique is within 2% on average. Moreover, using optimization, the overhead can be reduced at the cost of slightly higher error.

5.2 Motivating Example

We motivate the application of simulation based coverage computation to coverage extraction during emulation through a simple example. Consider the example Verilog code in Figure 5.1. This is a code snippet from module `usbf_pa` of a USB design downloaded from [43].

```
1. //variable declarations, assign statements, always blocks etc…..
2. always @(posedge clk)
3.       send_zero_length_r <= send_zero_length;
4. always @(posedge clk)
5.      if(reset)       state <= IDLE;
6.      else            state <= next_state;
7. always @(state or send_data or tx_ready or tx_valid_r or send_zero_length_r)
8. begin
9.      //assignments…
10.     case (state)
11.             IDLE:
12.             begin
13.                     if(send_zero_length_r && send_data) begin
14.                             tx_valid_d = 1'b1;
15.                             next_state = WAIT;
16.                             dsel = 1'b1;
17.                     end else if(send_data) begin
18.                             tx_valid_d = 1'b1;
19.                             next_state = DATA;
20.                             dsel = 1'b1;
21.                     end
22.             end
23.             DATA:  //more decisions and assignments….
24.             //Remaining cases
25.     endcase
26.end
```

Figure 5.1: Verilog RTL code snippet from module `usbf_pa` of a USB design downloaded from [43]. `clk`, `rst`, `send_data` are inputs and the remaining variables are internal variables.

Now suppose that this design is downloaded onto an FPGA platform and an emulation test is applied to it. Our aim is to find the statement coverage of the test, i.e. statements executed during this emulation run.

A naïve approach is to add logic for each statement to determine if it was ever executed and to add a bit to record it. Clearly, the approach is infeasible due to the prohibitive area overheads. However, if we identify some key points in the code so that execution of these points ensures execution of a set of statements, the overheads can be greatly reduced by only recording the execution of these points. We find that the conditional statements in the code can effectively serve as such key points.

For example, consider the decision node d for line 13: `if (send_zero_length_r && send_data)`. Since this condition depends on the condition on line 11 (`state==IDLE`), the full expression for decision node is represented as `(state==IDLE)&&(send_zero_length_r==1)&&(send_data==1)`. We add logic to record if the condition ever evaluates to true (d ever triggers) during an emulation run. Given that d triggered during the emulation run, we can infer the following:

(1) Since `state` had value IDLE, the statements on line 5 were executed, including the initial assignment to `state`. Although `send_zero_length_r` held value 1, it is not sufficient to conclude that the assignment on line 3 was executed, since the assigned value (input `send_zero_length`) is unknown in that case.

(2) Assignments on lines 14 through 16 were also executed since they are conditioned on d.

(3) Assignments on line 9, other continuous assignments and unconditional statements inside other always blocks (actual statements not shown for clarity) were also executed. The assignment on line 3 was also executed. This is because such statements are always executed, independent of the triggering of d.

Note that the sets of statements in (1), (2) and (3) above are similar to the backward, forward and ancillary cones introduced in Definition 1. Thus using algorithms similar to the correctness based coverage computation (Section 3.4), these sets for each decision node can be determined from static analysis prior

to emulation. The trigger information from the emulation run can then be cross referenced with the RTL code using these sets to estimate the statement coverage of the test.

Moreover, it can be seen that recording the triggering of one decision node can give us information about coverage of seven statements (excluding the ones in the ancillary cone). This gives an insight into the effectiveness of using the decision nodes as the key points to record coverage.

5.3 Methodology

Figure 5.2 illustrates the main steps in our methodology to extract code coverage from emulation and prototyping platforms. Given a Verilog RTL design, we parse the design files to construct Control Flow Graphs (CFGs) augmented with variable dependency information for each module (Section 5.3.1). We also extract conditional statements as *decision nodes* from the CFGs in this step, which are candidates for synthesis. The CFGs are then traversed to obtain the RTL lines *covered* by each decision node (Section 5.3.2). The optimization step involves identifying a subset of decision nodes that satisfies coverage and size constraints (Section 5.3.3). Finally, logic to record triggering of the optimized set of decision nodes is added to the design (Section 5.3.4). The design is synthesized on the target emulation platform and code coverage statistics are derived from the trigger information (Section 5.3.5).

5.3.1 Extended CFG Construction

We reuse the extended CFG framework described in Section 3.2 in this case. For the design under consideration, we construct extended CFGs for each of the constituent modules.

We extract the conditional statements in the RTL code in the form of decision nodes in the CFG. Recall that in a CFG, a *decision node* represents conditional statements, i.e. `if` and `case` statements in the source code.

We refer to the conditional expression for a decision node evaluating to true as *triggering* of the decision node. In general, the CFG can contain nested decision nodes. In order for a decision node to trigger, other decision nodes on the path

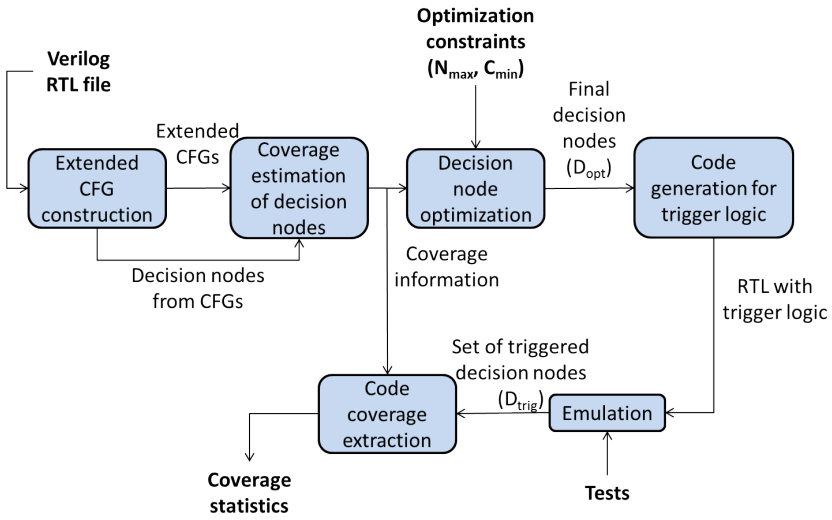


Figure 5.2: Block diagram describing the overall flow of our technique of coverage extraction from emulation.

from a top level node to that decision node need to trigger. Therefore, conditional expressions for such nodes are included in the representation of that decision node.

Figure 5.3 shows the CFG for the module shown from Figure 5.1. It consists of the CFGs for the three `always` processes starting at lines 2, 4 and 7.

It can be seen that each statement in the code is mapped to a node in the CFG. Nodes corresponding to `if` and `case` are decision nodes (shaded).

Although not shown in Figure 5.3, we map terminal `else` statements to separate decision nodes. The conditional expression for such a node is the negation of the conditional expression for the corresponding `if` decision node. For example, for the `always` process starting at line 4, the decision nodes are `d1: (reset==1)` and `d2: (reset==0)`.

5.3.2 Estimating Coverage of Decision Nodes

For each decision node, we find the set of statements such that during running of a test, the triggering of the decision node ensures execution of these statements. Note that this is similar to the result set of an assertion according to the simulation based coverage metric defined in Section 3.3.1. Therefore, we define *coverage of a decision node* on the same lines as Definition 1.

Definition 5 *(Decision node coverage) A statement s is covered by a decision node d if it belongs to one of the following categories.*

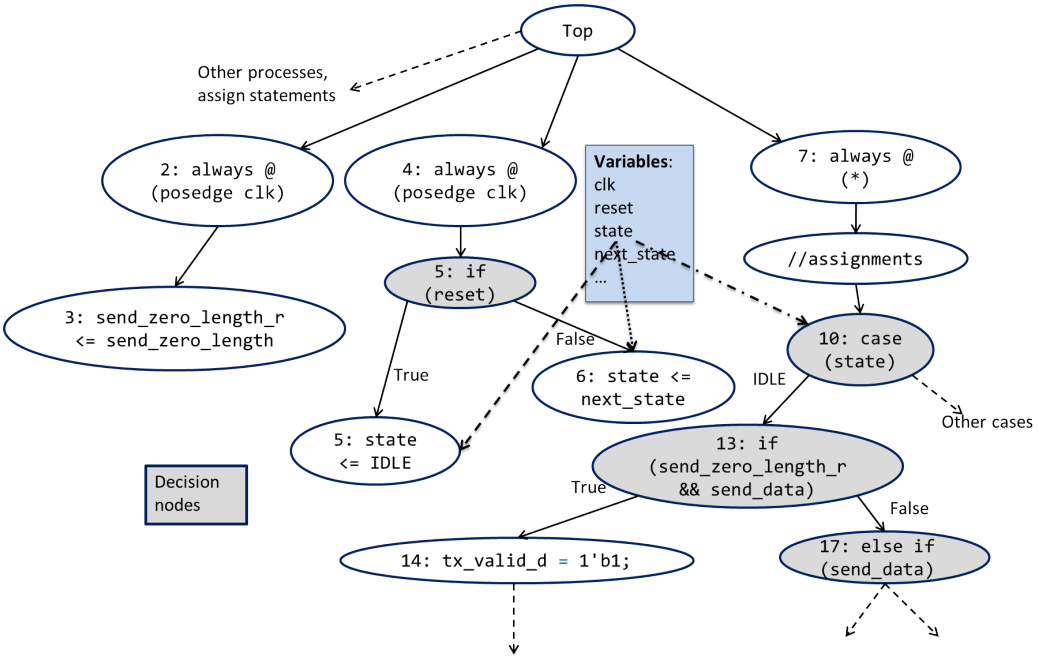


Figure 5.3: Control flow graph for the example RTL design from Figure 5.1 augmented with data flow information in the form of *assignments* (dashed line) and *decisions* (dash dot) and `initial assignment` (dotted line) for variables. For clarity, this information is shown only for the internal variable `state`.

(1) s *must be executed in order for the conditional expression for* d *to evaluate to true.*

(2) s *is executed because the conditional expression for* d *evaluates to true.*

(3) s *is executed irrespective of the evaluation of the conditional expression of* d*.*

(4) s *does not belong to the above 3 categories, but execution of a statement from (1) or (2) ensures the execution of* s*.*

We call the set of statements falling in categories (1) and (2) as the *backward cone* and the *forward cone* of the decision node, respectively. We use the term *ancillary cone* to denote the statements belonging to categories (3) and (4).

We attempt to express the conditional expression for a decision node as a proposition or conjunctions of propositions. If this is possible, the coverage of decision nodes is computed using the algorithms described in Section 3.4. Triggering of an assertion requires its antecedent to evaluate to true, while for a decision node

to trigger, the corresponding conditional expression must evaluate to true. Therefore, the backward and forward cone computation algorithms in Section 3.4 can be reused for decision nodes, where the conditional expression for the decision node is used in place of the antecedent.

We demonstrate our algorithm to find coverage of decision nodes in the example from Figure 5.1 and Figure 5.3. We again consider the decision node d for line 11: `if (send_zero_length_r && send_data)`. Since this is a nested condition, the expression for decision node is represented as `(state==IDLE) && (send_zero_length_r==1) && (send_data==1)`. Note that in this case, d is represented as a conjunction of propositions. Therefore, we use the algorithms in Section 3.4 for coverage computation.

Backward cone (B_d) computation: We run the $GetBackConeProposition(G, p)$ procedure with p: `(state==IDLE)` which is the only proposition in d involving internal variables. $GetMatchingAssignments(G, p)$ returns with the single initial assignment on line 5. Since exactly one matching assignment is found, we include the statement on line 5 in B_d. Also since the right hand side of the assignment is a constant, no recursive call is made on $GetBackConeProposition()$. We then traverse upwards starting from the CFG node mapped to line 5 and stop at the top level node. Running $GetBackConeProposition()$ with p: `(send_zero_length_r==1)` does not add more statements to B_d, since the assignment on line 3 assigns an unknown value (input `send_zero_length`) to `send_zero_length_r`.

Forward cone (F_d) computation: We identify the `always` process starting at line 7 as one containing decisions involving the variables in d. Starting at the top level node, we traverse downwards. The decision at line 10 can be evaluated using the proposition `(state==IDLE)` and we take the left edge. The decision at line 13 evaluates to true and therefore we take the left edge again. Statements on lines 10 and 13 visited thus far are included in F_d. The assignments on lines 14 as well as on 15 and 16 (not shown in the CFG) are also included in F_d. Forward cone computation stops at this point.

Ancillary cone (A_d) computation: In the first step, we include statements on line 9 (actual assignments not shown) and line 3. All continuous assignments in the module are also included in A_d in this step. In this case, the second step does not add any lines to A_d.

Figure 5.4 shows the CFG nodes included in B_d (blue), F_d (pink) and A_d (yellow) among the ones shown in Figure 5.3.

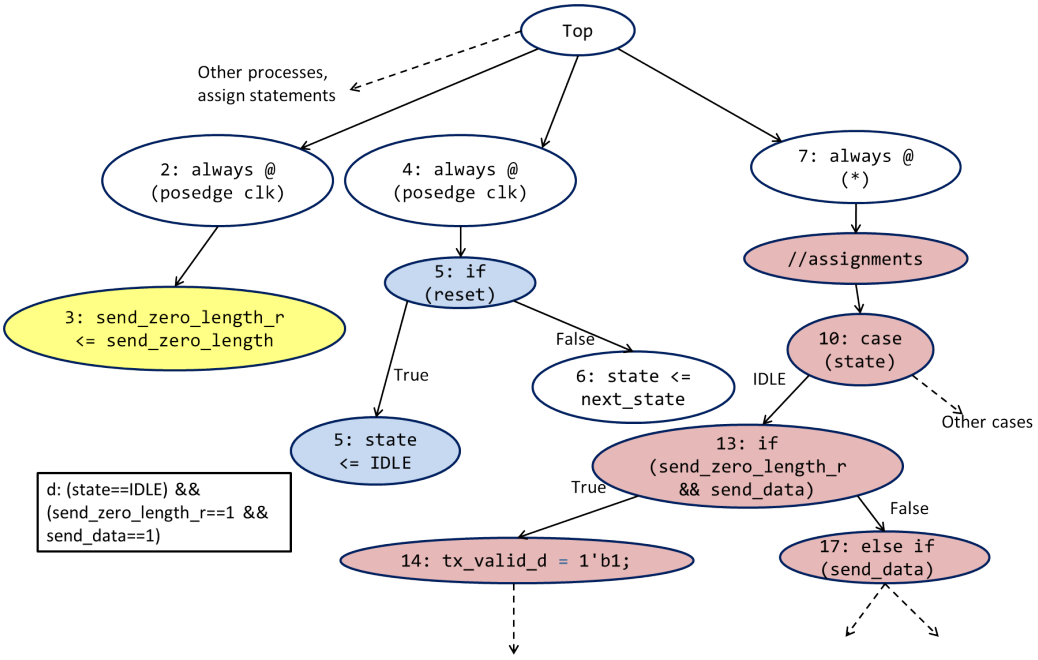


Figure 5.4: CFG from Figure 5.3 with nodes included in backward cone (blue), forward cone (pink) and ancillary cone (yellow) of d: (`state==IDLE && send_zero_length_r==1 && send_data==1`) shown.

If the conditional expression for a decision node is too complex to be expressed as a conjunction of propositions, we resort to simpler versions of the algorithms as follows.

Backward Cone It is not possible to compute the backward cone of a decision node if it is not expressed as a conjunction of propositions. In this case, we simply set it to $\varnothing$.

Forward Cone To compute the forward cone of a decision node in this case, we traverse the CFG downwards starting from the decision node until another decision node is encountered. The statements corresponding to the visited nodes are included in the forward cone.

Ancillary Cone Since the ancillary cone computation described in Section 3.4.3 is not based on propositions, it does not change in this case.

5.3.3 Optimization

It is possible to have multiple decision nodes covering the same statement. To compute coverage accurately, we require at least one decision node covering each statement. Moreover, synthesizing the trigger logic for decision nodes incurs area costs. Therefore we perform optimization to obtain a subset of the decision nodes that satisfy certain user defined constraints. Specifically, we find a subset of decision nodes with maximum total coverage that satisfies the following constraints:

1. Total number of decision nodes $\leq N_{max}$.

2. Fraction of total lines covered in the RTL code for each module $\geq C_{min}$.

Here N_{max} and C_{min} are user-specified.

This problem is identified as a variant of the *Maximum coverage problem* and is formulated as an Integer Linear Program (ILP) [90]. This ILP is solved to get the final set of decision nodes, denoted by D_{opt}.

Let $M_0, M_1, \ldots$ be the modules that constitute the target RTL design and L_{M_0}, $L_{M_1}, \ldots$ the sets of lines in the RTL code for these modules, respectively. Then $L = L_{M_0} \cup L_{M_1} \cup \ldots = \{l_0, l_1, \ldots\}$ is the set of all lines in the design.

Let $D = \{d_0, d_1, \ldots\}$ be the set of all decision nodes and $L_{d_0}, L_{d_1}, \ldots$ be the sets of RTL lines covered by these decision nodes, respectively.

Our goal is to find the set $D_{opt} \subseteq D$ that covers the set of lines $L_{opt} \subseteq L$ and satisfies the conditions mentioned above.

We define two sets of binary variables, viz. $\{x_0, x_1, \ldots\}$ and $\{y_0, y_1, \ldots\}$. $x_i = 1$ if $d_i \in D_{opt}$ and 0 otherwise. Similarly, $y_j = 1$ if $l_j \in L_{opt}$ and 0 otherwise. The ILP is then formulated as follows.

Maximize $\sum y_j$

subject to:

$\sum x_i \leq N_{max}$;

$\forall l_j, \sum_{l_j \in L_{d_i}} x_i \geq y_j$;

$\forall M_i, |L_{M_i} \cap L_{opt}| \geq C_{min} \times |L_{M_i}|$;

$\forall i, x_i \in \{0, 1\}$;

$\forall j, y_j \in \{0, 1\}$;

We use the MATLAB solver `bintprog()` to solve this ILP.

Note that when $C_{min} < 1$ is used, there could be statements in the code not covered by any decision node in the D_{opt}. As a result, the coverage information

for such statements extracted from emulation may not be accurate. However, fewer decision nodes may be required in this case, which reduces the area cost. This trade-off between accuracy of coverage results and area cost is demonstrated by the case study in Section 5.4.

5.3.4 Decision Node Synthesis

We add a variable `cov_di` to record the triggering of each decision node $d_i \in D_{opt}$. The RTL code added is shown in Figure 5.5. When reset is asserted, `cov_di` is initialized to 0. When the condition corresponding to d_i evaluates to true during running of a test, `cov_di` is set to 1. Thus at the end of the test, the value of `cov_di` denotes if the decision node d_i was ever triggered. The value must be captured before subsequent resets since a reset clears `cov_di`.

```
always @ (posedge clk)
  if (reset)
    cov_di <= 1'b0;
  else if (cond_di)
    cov_di <= 1'b1;
```

Figure 5.5: RTL code to record triggering of decision node d_i with conditional expression `cond_di`.

5.3.5 Coverage Extraction from Emulation

The target design is synthesized with the additional trigger logic for decision nodes and loaded onto an FPGA. The additional logic consists of a flip-flop for `cov_di` along with some combinational logic for the condition in d_i. An emulation test is run on the FPGA, and at the end the values of `cov_di` flip-flops are extracted.

The subset $D_{trig} = \{d_i \in D_{opt} | \texttt{cov\_di} = 1\}$ gives the set of decision nodes that triggered during the test. Let S_{d_i} be the set of statements covered by decision node d_i, which is already available from the coverage estimation step. Therefore, we can find the set $S_{cov} = \bigcup_{d_i \in D_{trig}} S_{d_i}$ of statements that were definitely executed during the test. Let S_{sim} be the set of statements reported as covered by a simulator

for the same test. Note that S_{cov} may not include all the lines in S_{sim}, but is guaranteed to be a conservative estimate, or in other words $S_{cov} \subseteq S_{sim}$.

From S_{cov}, the standard code coverage metric of statement coverage, i.e. fraction of RTL statements executed, can be obtained directly. Moreover, since each decision node corresponds to a condition or *branch* in the RTL code, branch coverage can also be derived from D_{trig}.

5.4 Case Study: An Industrial Design

We evaluated our methodology using an industrial RTL design. We considered a signal processing module that generates timing recovery strobes for a receiver front end. The sub-blocks included state machines, counters and math computations.

Experiments were carried out on an Intel Xeon Quad Core @ 3.16 GHz with 32 GB RAM running Red Hat Linux v2.6. Starting from the design files, generating the trigger logic for final decision nodes completed within 30 seconds.

We used MATLAB solver `bintprog()` for optimization. Xilinx FPGA platform was used to emulate the original design as well as the modified design with the additional trigger logic for decision nodes (Section 5.3.4). The values of `cov_di` flip-flops were extracted using the Xilinx ChipScope tool. For comparison of code coverage statistics, we used the ModelSim RTL simulator.

5.4.1 Decision Node Generation and Optimization

Table 5.1 shows the number of lines, number of total decision nodes and number of decision nodes after optimization for the 6 RTL modules M_1 through M_6 belonging to the design block under consideration. Column 3 denotes the number of lines actually considered for coverage. This includes lines of code containing assignments and conditional statements and excludes blank lines, comments, variable declarations, module instantiation and port connections, etc. Note that modules with sequential code are considered, since we need the clock and reset signals present only in sequential modules for trigger logic.

Table 5.1 shows the code size and number of decision nodes for the modules under consideration.

Table 5.1: Number of lines and decision nodes before and after optimization for each RTL module. Column 3 denotes the number of statements considered for coverage. Decision node optimization is performed for two configurations (1) $C_{min} = 1$ and (2) $C_{min} = 0.9$, $N_{max} = 256$.

Module	Lines	Statements for coverage	Total decision nodes	Decision nodes for config. (1)	Decision nodes for config. (2)
M_1	1855	927	177	177	85
M_2	271	3	2	2	2
M_3	254	85	40	39	38
M_4	195	50	15	15	10
M_5	588	210	108	95	80
M_6	907	255	62	59	26

We ran decision node optimization with two configurations (Columns 4 and 5 in Table 5.1). In configuration (1), $C_{min} = 1$ was used, which ensures that all statements are covered by D_{opt}. The total number of decision nodes in this case was 411. This was done in order to prune out redundant decision nodes while keeping the coverage target to 100%. In optimization configuration (2), we relaxed the per module coverage constraint to 90% ($C_{min} = 0.9$) and bounded the total number of decision nodes by a number smaller than 411 ($N_{max} = 256$).

Optimization for configuration (1) reduced the number of decision nodes only for M_3, M_5 and M_6. For remaining modules, there were no redundant decision nodes. In other words, each decision node covered a statement not covered by any other decision node. Configuration (2) showed reduction in the number of decision nodes in nearly all modules. The reduction is more significant in M_1 and M_6. This is because they contain decision nodes that add very little extra coverage, and hence the 90% coverage target can be met even after removing those.

5.4.2 FPGA Overheads

Table 5.2 summarizes the FPGA usage overheads of trigger logic for both optimization configurations. The usage is expressed in terms of lookup table (LUT) and flip-flop (FD_LD) counts. Changes in the usage of the remaining FPGA primitives were insignificant. The FD_LD overhead mainly corresponds to the actual `cov_di` flip-flops to store trigger information. The overhead in LUT counts is due to additional combinational logic to record the triggering.

Table 5.2: Percentage FPGA usage overheads of trigger logic for the set of decision nodes optimized for configurations (1) $C_{min} = 1$ and (2) $C_{min} = 0.9$, $N_{max} = 256$.

Usage primitive	Overhead for config. (1)	Overhead for config. (2)
LUT	3.28%	2.71%
FD_LD	9.21%	5.36%

We find that the overheads are within 10% in all cases. Moreover, the overheads for configuration (2) are up to 42% lower than (1), which shows how decision node optimization can help lower overheads. As expected, the FD_LD overhead is found to be roughly proportional to the number of decision nodes for which trigger information is being recorded (411 and 256 for the two configurations).

Table 5.1 shows that for most modules, the number of decision nodes is much smaller (up to 5x for M_1) than the number of statements. This illustrates the benefit of static analysis to identify coverage extraction logic. In the absence of such analysis, logic to record triggering of *each statement* would be required for coverage extraction. Since we map decision nodes to statements and record triggering of decision nodes alone, we can extract the same coverage information at much lower FPGA area overheads.

5.4.3 Code Coverage Comparison with Simulator

We ran a system level test (Test 1) on the modified designs corresponding to configurations (1) and (2) and collected trigger information for all decision nodes. The trigger information was mapped back to the source code to determine statements executed during the test. The same test on the original design was run in a simulator, and code coverage information was obtained from the simulator. For modified design for configuration (2), the process was repeated for a much longer system level test (Test 2).

Comparison between the statement coverage reported from emulation and by the simulator revealed that, in all cases, coverage obtained from emulation is indeed a conservative estimate of the actual coverage from simulation. We report the number of statements that were covered according to the simulator but reported as *not covered* by our technique as a percentage of total number of statements.

We report detailed results for Test 1 in Table 5.3. When decision nodes with optimization configuration (1) were used, the average error was only 1.8%. A

Table 5.3: Percentage error in line coverage extracted from FPGA using decision nodes for optimization configurations (1) and (2) for Test 1 and configuration (2) for Test 2. The error represents the number of covered statements *not* reported using our method as a percentage of number of statements (Column 3 in Table 5.1).

	Simulation runtime	Emulation runtime	Module	Error for config. (1)	Error for config. (2)
Test 1	3 hours	30 seconds	M_1	0.22%	0.86%
			M_2	0%	0%
			M_3	4.71%	5.88%
			M_4	4%	4%
			M_5	2.86%	2.86%
			M_6	0.78%	7.06%
Test 2	20 hours	5 minutes	Average	-	3.80%

more detailed analysis showed that the discrepancy between coverages can be mainly attributed to some `default` cases of combinational `case` statements. The `cov_di` flip-flops require a reset in order to start recording triggering of decision nodes. Therefore, for the default cases that were hit before reset was asserted, the corresponding flip-flops failed to record triggering.

As expected, the error in reported coverage was found to be slightly higher when a lower number of decision nodes were used. Table 5.1 shows that for optimization configuration (2), the number of decision nodes was significantly lower compared to configuration (1) for modules M_1 and M_6. Table 5.3 shows that this resulted in around three and nine times increase in error in reported coverage. However, for other modules, the increase in error was less significant. Therefore, the average error increased only to 3.8% from 1.8%. From this observation, we can conclude that our technique can report meaningful code coverage estimates with reduced overheads.

We observed similar coverage statistics for Test 2. In Table 5.3, we only report the average error in coverage across all modules in this case.

Note that running a test in emulation is about 300x faster than simulation. In particular, Test 2 has a prohibitively large runtime of 20 hours during simulation. However, we are able to obtain coverage statistics within 4% error from emulation for this test as well, thus illustrating the key benefit of our technique.

CHAPTER 6

RESOURCES

This chapter gives instructions on obtaining and using our tools for code coverage and fault coverage analysis of assertions. Sources for the tools are available at:

`http://users.crhc.illinois.edu/athaval1`

Both tools run on Linux and require `perl` and `gcc/g++`. The code coverage tool requires `flex` and `bison` for Verilog parsing. The fault coverage tool requires the formal verifier Cadence IFV. The tools have been tested on `Debian Linux 2.6.32-5-amd64` running `perl v5.8.7, gcc/g++ v4.4.5, flex v2.5.35, GNU Bison v2.4.1` and `IFV v08.20-s021`.

6.1 Using the Code Coverage Tool

Follow these steps to run the code coverage tool:

1. Download the source from:
   `http://users.crhc.illinois.edu/athaval1`
   `/code_coverage.tar.gz`.

2. In a terminal, navigate to the download directory and type:
   `$tar -zxvf code_coverage.tar.gz`
   This creates a directory called `code_coverage`. Navigate to this directory.

3. To compile, type:
   `$make`

4. For a quick run, type:
   `$perl ccover.pl -metric=fv example.v example.assert`

This step runs correctness based coverage analysis for the assertion in file `example.assert` for the design in `example.v` and prints the covered statements to the terminal.

5. For detailed instructions and more options, please see the `README` file in the source directory.

6.2 Using the Fault Coverage Tool

Follow these steps to run the fault coverage tool:

1. Download the source from:
   `http://users.crhc.illinois.edu/athaval1/fault_coverage.tar.gz`.

2. In a terminal, navigate to the download directory and type:
   `$tar -zxvf fault_coverage.tar.gz`
   This creates a directory called `fault_coverage`. Navigate to this directory.

3. To compile, type:
   `$make`

4. For a quick run, type:
   `$perl fcover.pl example.v example.assert`
   This step first instruments the design source in `example.v` so that SEU faults can be injected at each state variable. The instrumented source is then used to inject faults in state variables one at a time to determine if the assertion in file `example.assert` detects the fault. Results showing faults detected by the assertion are printed to the terminal.

5. For detailed instructions and more options, please see the `README` file in the source directory.

CHAPTER 7

CONCLUSIONS

The work presented in this thesis attempts to extend the coverage analysis used for simulation based verification to two other major components of pre-silicon verification: assertion based verification and emulation and prototyping.

Code coverage metrics are defined for each of the two use cases of assertions, viz. simulation and formal verification. Coverage analysis is performed through a direct analysis of the CFG constructed from an RTL design as opposed to previous approaches that either use the state transition graph [32, 33] or an indirect mutation analysis [18]. The same CFG framework is reused to identify key recording points in the design to increase observability and enable efficient code coverage estimation during emulation and prototyping. A formal technique to rank the automatically generated assertions on the basis of fault detection is also presented.

Experiments with an industrial design and open source designs from OpenCores [43] demonstrate the effectiveness of the proposed techniques. For USB 2.0 and OpenRISC open source designs, our code coverage analysis technique for assertions effectively finds the statements truly relevant to the correctness of an assertion. For an industrial design, our technique for coverage extraction emulation reports code coverage within 2% error around 10% FPGA usage overhead. Experiments with SpaceWire protocol [88] design show that our fault detection based ranking technique for GoldMine [40] identifies 4% high-quality assertions that detect $>$30% of the injected faults.

This work is only an initial step towards a unified coverage analysis for pre-silicon verification. As with the statement coverage metric, coverage analysis for other coverage metrics is needed in the context of assertion and emulation based verification. Such an approach is necessary to assess the overall progress of the various verification activities and identify parts of the design that need further attention, be it in the form of a more comprehensive test suite or high- quality assertions. Together, these improvements will help achieve coverage closure and ensure correctness of the design before synthesizing it onto costly silicon.

REFERENCES

[1] A. Evans, A. Silburt, G. Vrckovnik, T. Brown, M. Dufresne, G. Hall, T. Ho, and Y. Liu, "Functional verification of large ASICs," in *Proceedings of the 35th Annual Design Automation Conference*, 1998, pp. 650–655.

[2] *IEEE Verilog Hardware Description Language*, IEEE Standard 1364, 2001.

[3] *IEEE VHDL Language Reference Manual*, IEEE Standard 1076, 2009.

[4] S. Taylor, M. Quinn, D. Brown, N. Dohm, S. Hildebrandt, J. Huggins, and C. Famey, "Functional verification of a multiple-issue, out-of-order, superscalar Alpha processor-the DEC Alpha 21264 microprocessor," in *Proceedings of the 35th Annual Design Automation Conference*, 1998, pp. 638–643.

[5] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Des. Test*, vol. 18, no. 4, pp. 36–45, July 2001.

[6] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design*, 2nd ed. Norwell, MA: Kluwer Academic Publishers, 2004.

[7] H. Foster, *Applied Assertion-Based Verification: An Industry Perspective*. Hanover, MA: Now Publishers Inc., 2009.

[8] M. Boule, J.-S. Chenard, and Z. Zilic, "Assertion checkers in verification, silicon debug and in-field diagnosis," in *Proceedings of the Eighth International Symposium on Quality Electronic Design*, 2007, pp. 613–620.

[9] *IEEE Standard for Property Specification Language (PSL)*, IEEE Standard 1850, 2005.

[10] *IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language*, IEEE Standard 1800, 2009.

[11] A. Gupta, "Assertion-based verification turns the corner," *IEEE Des. Test*, vol. 19, no. 4, pp. 131–132, July 2002.

[12] K. Oner, L. Barroso, S. Iman, J. Jeong, K. Ramamurthy, and M. Dubois, "The design of RPM: An FPGA-based multiprocessor emulator," in *Proceedings of the Third International Symposium on Field Programmable Gate Arrays*, 1995, pp. 60–66.

[13] J. Ray and J. C. Hoe, "High-level modeling and FPGA prototyping of microprocessors," in *Proceedings of the 11th International Symposium on Field Programmable Gate Arrays*, 2003, pp. 100–107.

[14] J. Lach, W. Mangione-Smith, and M. Potkonjak, "Efficient error detection, localization, and correction for FPGA-based debugging," in *Proceedings of the 37th Annual Design Automation Conference*, 2000, pp. 207–212.

[15] G. Ganapathy, R. Narayan, C. Jorden, M. Wang, and J. Nishimura, "Hardware emulation for functional verification of K5," in *Proceedings of the 33rd Annual Design Automation Conference*, 1996, pp. 315–318.

[16] J. Gateley et al., "UltraSPARC-I emulation," in *Proceedings of the 32nd Annual Design Automation Conference*, 1995, pp. 13–18.

[17] M. Gschwind, V. Salapura, and D. Maurer, "FPGA prototyping of a RISC processor core for embedded applications," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 9, no. 2, pp. 241–250, April 2001.

[18] H. Chockler, O. Kupferman, and M. Vardi, "Coverage metrics for formal verification," *Int. J. Softw. Tools Technol. Transf.*, vol. 8, no. 4, pp. 373–386, August 2006.

[19] A. Piziali, *Functional Verification Coverage Measurement and Analysis*. Norwell, MA: Kluwer Academic Publishers, 2004.

[20] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, December 1997.

[21] M. Kantrowitz and L. Noack, "I'm done simulating; now what? Verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor," in *Proceedings of the 33rd Annual Design Automation Conference*, 1996, pp. 325–330.

[22] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets, "A study in coverage-driven test generation," in *Proceedings of the 36th Design Automation Conference*, 1999, pp. 970–975.

[23] Y. Hoskote, D. Moundanos, and J. Abraham, "Automatic extraction of the control flow machine and application to evaluating coverage of verification vectors," in *Proceedings of the 1995 International Conference on Computer Design: VLSI in Computers and Processors*, 1995, pp. 532–537.

[24] L. Fournier, A. Koyfman, and L. Levinger, "Developing an architecture validation suite: Application to the PowerPC architecture," in *Proceedings of the 36th Annual Design Automation Conference*, 1999, pp. 189–194.

[25] H. Foster and P. Yeung, "Planning formal verification closure," in *Proceedings of the IEC Design Conference*, 2007. [Online]. Available: http://www.iec.org/newsletter/jan08_1/tech_briefing_foster.pdf

[26] S. Devadas, A. Ghosh, and K. Keutzer, "An observability-based code coverage metric for functional simulation," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 1996, pp. 418–425.

[27] F. Fallah, S. Devadas, and K. Kuetzer, "OCCOM: Efficient computation of observability-based code coverage metrics for functional verification," in *Proceedings of the 35th Annual Design Automation Conference*, 1998, pp. 152–157.

[28] D. V. Campenhout, H. Al-Asaad, J. P. Hayes, T. Mudge, and R. B. Brown, "High-level design verification of microprocessors via error modeling," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 3, no. 4, pp. 581–599, October 1998.

[29] T. A. Budd and A. S. Gopal, "Program testing by specification mutation," *Comput. Lang.*, vol. 10, no. 1, pp. 63–73, January 1985.

[30] D. Moundanos, J. Abraham, and Y. Heskote, "A unified framework for design validation and manufacturing test," in *Proceedings of the IEEE International Test Conference on Test and Design Validity*, 1996, pp. 875–884.

[31] J.-Y. Jou and C.-N. J. Liu, "Coverage analysis techniques for HDL design validation," in *Proceedings of the 6th Asia Pacific Conference on cHip Design Languages*, 1999, pp. 48–55.

[32] Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao, "Coverage estimation for symbolic model checking," in *Proceedings of the 36th Annual Design Automation Conference*, 1999, pp. 300–305.

[33] S. Katz, O. Grumberg, and D. Geist, ""Have I written enough properties?" - A method of comparison between specification and implementation," in *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 1999, pp. 280–297.

[34] H. Chockler, O. Kupferman, and M. Y. Vardi, "Coverage metrics for temporal logic model checking," *Form. Methods Syst. Des.*, vol. 28, no. 3, pp. 189–212, May 2006.

[35] H. Chockler, O. Kupferman, R. P. Kurshan, and M. Y. Vardi, "A practical approach to coverage in model checking," in *Proceedings of the 13th International Conference on Computer Aided Verification*, 2001, pp. 66–78.

[36] O. Kupferman, W. Li, and S. Seshia, "A theory of mutations with applications to vacuity, coverage, and fault tolerance," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, 2008, pp. 1–9.

[37] A. Fedeli, F. Fummi, and G. Pravadelli, "Properties incompleteness evaluation by functional verification," *IEEE Trans. Comput.*, vol. 56, no. 4, pp. 528–544, April 2007.

[38] V. Singhal and P. Aggarwal, "Using coverage to deploy formal verification in a simulation world," in *Proceedings of the 23rd international conference on Computer aided verification*, 2011, pp. 44–49.

[39] P. Aggarwal, D. Chu, V. Kadamby, and V. Singhal, "Planning for end-to-end formal using simulation-based coverage," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (Invited Tutorial)*, 2010, pp. 2183–2187.

[40] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, "GoldMine: Automatic assertion generation using data mining and static analysis," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2010, pp. 626–629.

[41] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum, "Program slicing for VHDL," *Int. J. Softw. Tools Technol. Transf.*, vol. 4, no. 1, pp. 125–137, 2002.

[42] S. Vasudevan, E. A. Emerson, and J. A. Abraham, "Improved verification of hardware designs through antecedent conditioned slicing," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 1, pp. 89–101, February 2007.

[43] OpenCores website, 2011. [Online]. Available: http://www.opencores.org/

[44] L. Liu, D. Sheridan, W. Tuohy, and S. Vasudevan, "Towards coverage closure: Using GoldMine assertions for generating design validation stimulus," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2011, pp. 1–6.

[45] T.-H. Wang and C. G. Tan, "Practical code coverage for Verilog," in *Proceedings of the Fourth IEEE International Verilog HDL Conference*, 1995, pp. 99–104.

[46] J. Shen and J. A. Abraham, "An RTL abstraction technique for processor microarchitecture validation and test generation," *J. Electron. Test.*, vol. 16, no. 1-2, pp. 67–81, February 2000.

[47] J. Shen and J. Abraham, "Verification of processor microarchitectures," in *Proceedings of the 17th IEEE VLSI Test Symposium*, 1999, pp. 189–194.

[48] S. Ur and Y. Yadin, "Micro architecture coverage directed generation of test programs," in *Proceedings of the 36th Annual Design Automation Conference*, 1999, pp. 175–180.

[49] VCS datasheet, 2011, Synopsys, Inc., Mountain View, CA. [Online]. Available: http://www.synopsys.com/Tools/Verification/FunctionalVerification/Documents/vcs-ds.pdf

[50] Incisive Enterprise Verifier datasheet, 2010, Cadence Design Systems, Inc., San Jose, CA. [Online]. Available: http://www.cadence.com/rl/Resources/datasheets/incisive_enterprise_verifier_ds.pdf

[51] ModelSim datasheet, 2008, Mentor Graphics Corporation, Wilsonville, OR. [Online]. Available: http://www.mentor.com/products/fv/modelsim/upload/datasheet.pdf

[52] S. Ur and A. Ziv, "Off-the-shelf vs. custom made coverage models, which is the one for you?" in *Proceedings of the 7th International Conference on Software Testing, Analysis, and Review*, 1998.

[53] D. A. Peled, D. Gries, and F. B. Schneider, Eds., *Software Reliability Methods*. Secaucus, NJ: Springer-Verlag, 2001.

[54] V. Agrawal, S. Seth, and P. Agrawal, "Fault coverage requirement in production testing of LSI circuits," *IEEE J. Solid-State Circuits*, vol. 17, no. 1, pp. 57–61, February 1982.

[55] C. Stoucny, R. Davies, P. McKernan, and T. Truong, "Alpha 21164 manufacturing test development and coverage analysis," *IEEE Des. Test Comput*, vol. 15, no. 3, pp. 98–104, July-September 1998.

[56] R. Guo, S. Mitra, E. Amyeen, J. Lee, S. Sivaraj, and S. Venkataraman, "Evaluation of test metrics: Stuck-at, bridge coverage estimate and gate exhaustive," in *Proceedings of the 24th IEEE VLSI Test Symposium*, 2006, pp. 66–71.

[57] A. Turing, "Checking a large routine," in *Report of a Conference on High Speed Automatic Calculating machines*, 1949, pp. 67–69.

[58] R. W. Floyd, "Assigning meanings to programs," in *Proceedings of the Symposia on Applied Mathematics*, 1967, pp. 19–32.

[59] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 26, no. 1, pp. 53–56, January 1983.

[60] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, April 1986.

[61] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th International Conference on Software Engineering*, 2002, pp. 291–301.

[62] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 99–123, February 2001.

[63] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 4–16.

[64] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: Dynamic symbolic execution for invariant inference," in *Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 281–290.

[65] S. Hangal, S. Narayanan, N. Chandra, and S. Chakravorty, "IODINE: A tool to automatically infer dynamic invariants for hardware designs," in *Proceedings of the 42nd Annual Design Automation Conference*, 2005, pp. 775–778.

[66] P.-H. Chang and L.-C. Wang, "Automatic assertion extraction via sequential data mining of simulation traces," in *Proceedings of the 15th Asia and South Pacific Design Automation Conference*, 2010, pp. 607–612.

[67] W. Li, A. Forin, and S. Seshia, "Scalable specification mining for verification and diagnosis," in *Proceedings of the 47th Annual Design Automation Conference*, 2010, pp. 755–760.

[68] J. Misra, "Prospects and limitations of automatic assertion generation for loop programs," *SIAM J. Comput.*, vol. 6, no. 4, pp. 718–729, 1977.

[69] A. Hazra, A. Banerjee, S. Mitra, P. Dasgupta, P. P. Chakrabarti, and C. R. Mohan, "Cohesive coverage management for simulation and formal property verification," in *Proceedings of the 2008 IEEE Computer Society Annual Symposium on VLSI*, 2008, pp. 251–256.

[70] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[71] S. A. Seshia, W. Li, and S. Mitra, "Verification-guided soft error resilience," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2007, pp. 1442–1447.

[72] U. Krautz, M. Pflanz, C. Jacobi, H. Tast, K. Weber, and H. Vierhaus, "Evaluating coverage of error detection logic for soft errors using formal methods," in *Proceedings of the Conference on Design, Automation and Test in Europe*, vol. 1, 2006, pp. 1–6.

[73] H. Asadi and M. Tahoori, "Soft error modeling and protection for sequential elements," in *Proceedings of the 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2005, pp. 463–471.

[74] A. Maheshwari, I. Koren, and N. Burleson, "Techniques for transient fault sensitivity analysis and reduction in VLSI circuits," in *Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2003, pp. 597–604.

[75] K. Mohanram and N. Touba, "Cost-effective approach for reducing soft error failure rate in logic circuits," in *Proceedings of the International Test Conference*, 2003, pp. 893–901.

[76] H. Nguyen and Y. Yagil, "A systematic approach to SER estimation and solutions," in *Proceedings of the 41st Annual International Reliability Physics Symposium*, 2003, pp. 60–70.

[77] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003, pp. 29–40.

[78] N. Wang, J. Quek, T. Rafacz, and S. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2004, pp. 61–70.

[79] VN-Cover Emulator Product Overview, TransEDA, Los Gatos, CA, 2005. [Online]. Available: http://www.transeda.com/site/images/stories/vn-cover-emulator.pdf

[80] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 27, no. 7, pp. 1165–1178, July 2008.

[81] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, July 1987.

[82] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE Softw.*, vol. 25, no. 5, pp. 22–29, September 2008.

[83] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security and Privacy*, vol. 2, no. 6, pp. 76–79, November 2004.

[84] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. 10, no. 4, pp. 352–357, July 1984.

[85] W. Wang et al., "A comprehensive high-level synthesis system for control-flow intensive behaviors," in *Proceedings of the 13th ACM Great Lakes Symposium on VLSI*, 2003, pp. 11–14.

[86] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, 1977, pp. 46–57.

[87] R. Baumann, "Soft errors in advanced computer systems," *IEEE Des. Test*, vol. 22, no. 3, pp. 258–266, May 2005.

[88] "SpaceWire-links, nodes, routers and networks (ECSS-E-ST-50-12C)," July 2008, European Corporation for Space Standardization. [Online]. Available: http://spacewire.esa.int/content/Standard/ECSS-E50-12A.php

[89] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Secaucus, NJ: Springer-Verlag, 2006.

[90] V. V. Vazirani, *Approximation Algorithms*. Secaucus, NJ: Springer-Verlag, 2001.